第五次作业 微调中文预训练模型进行文本分类

杜晋华 2022310811

要求：在智源指数 CUGE（http://cuge.baai.ac.cn/#/）任务上微调中文预训练模型。
目标：借此任务熟悉利用预训练模型的过程，感受预训练模型的能力。
作业结构：

一、CCPM 语言理解能力-词句级 | 古诗文识记

- 通过编写代码在 CCPM 的训练集上微调一个 thu-cbert-character 模型（中文单字模型，即该模型以汉字为 token）
- 利用验证集（valid.jsonl）进行checkpoint 选择和超参调整
- 最终在测试集（test_public.jsonl）上进行预测
- 在智源指数 CUGE 提交评测上提交 test_public 上的预测结果，根据实验过程和结果编写实验报告。

1. 代码编写

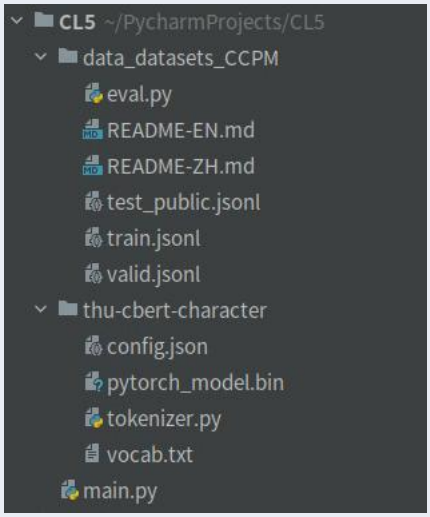
Step1：下载数据集 CCPM

- ✓ CCPM 给定中国古典诗歌的现代文描述，要求从候选的四句诗中挑选出与给定的现代文描述意思符合的那一句诗歌。（http://cuge.baai.ac.cn/#/dataset?id=1&name=CCPM）
- ✓ 训练集：21,778 句；验证集：2,720 句；测试集：2,720 句。
- ✓ 每条数据包含诗歌对应的翻译(translation，以字符串形式存储)，四个对应诗歌的备选选项(choice，以列表形式存储)，答案编号(answer，为 0-3 之间的整数)。

```
{
  "translation": "一生当中疾病缠身今日独上高台。",
  "choices": ["一春多病几登台", "百年多病独登台", "百年多病负登临", "况多愁病独登台"],
  "answer": 1
}
```


Step2：下载 thu-cbert-character 模型代码，并构建 Python 项目

构建后项目的整体结构如下：

Pycharm 截图	一级目录	二级目录	文件	文件用途
	CL5	data_datasets_CCPM	eval.py	代码，用于模型测试评估
			test_public.jsonl	验证用数据集
			train.jsonl	训练用数据集
			test.jsonl	测试用数据集
		Thu-cbert-character	config.json	配置，项目涉及超参
			pytorch_model.bin	已经预训练好的模型文件
			tokenizer.py	用于对 vocab 词典进行取 token
			vocab.txt	vocab 词典
		/	main.py	核心代码，完成本任务

项目环境配置为：CUDA11.6　Python3.10
需要额外依赖包：datasets　jsonlines　scikit-learn　transformers

Step3：书写 main 代码

✓ **加载数据集**

```
ccpm_train = load_dataset(path="./data_datasets_CCPM/", data_files="train.jsonl")
ccpm_valid = load_dataset(path="./data_datasets_CCPM/", data_files="valid.jsonl")
ccpm_test = load_dataset(path="./data_datasets_CCPM/", data_files="test_public.jsonl")
```

测试结果：

```
# 测试代码
ccpm_train["train"][0]
```

```
{'translation': '诗人啊，你竟像在遥远的地方站立船头。',
 'choices': ['行人初上木兰舟', '骚人遥驻木兰舟', '有人独上木兰舟', '行人迢递木兰舟'],
 'answer': 1}
```

✓ **数据集分裂**

数据集原始状态每条数据如下：

```
{
  "translation": "一生当中疾病缠身今日独上高台。",
  "choices": ["一春多病几登台", "百年多病独登台", "百年多病负登临", "况多愁病独登台"],
  "answer": 1
}
```

将其转化为 translation 与每个 choice 成对的数据，方便模型进行学习，例如：

```
{
    "translation": "一生当中疾病缠身今日独上高台。",
    "choice": "一春多病几登台",
    "label": 0
}
{
    "translation": "一生当中疾病缠身今日独上高台。",
    "choice": "百年多病独登台",
    "label": 1
}
{
    "translation": "一生当中疾病缠身今日独上高台。",
    "choice": "百年多病负登临",
    "label": 0
}
{
    "translation": "一生当中疾病缠身今日独上高台。",
    "choice": "况多愁病独登台",
    "label": 0
}
```

在评测时，可以比较对于同一个 translation 的几条不同数据，选择对 label=1 预测分数最大的作为答案。

实现代码：

```
# 数据集分裂处理
ccpm_train_split = []
for i in range(0, len(ccpm_train["train"])):
    for j in range(0, 4):
        temp_json = {"translation": ccpm_train["train"][i]["translation"]}
        temp_json["choices"] = ccpm_train["train"][i]["choices"][j]
        if j == int(ccpm_train["train"][i]["answer"]):
            temp_json["answer"] = 1
        else:
            temp_json["answer"] = 0
        ccpm_train_split.append(temp_json)

ccpm_valid_split = []
for i in range(0, len(ccpm_valid["train"])):
    for j in range(0, 4):
        temp_json = {"translation": ccpm_valid["train"][i]["translation"]}
        temp_json["choices"] = ccpm_valid["train"][i]["choices"][j]
        if j == int(ccpm_valid["train"][i]["answer"]):
            temp_json["answer"] = 1
        else:
            temp_json["answer"] = 0
        ccpm_valid_split.append(temp_json)

ccpm_test_split = []
for i in range(0, len(ccpm_test["train"])):
    for j in range(0, 4):
        temp_json = {"translation": ccpm_test["train"][i]["translation"]}
        temp_json["choices"] = ccpm_test["train"][i]["choices"][j]
        temp_json["answer"] = 0
        ccpm_train_split.append(temp_json)
```

（上面截图的最后一行，_train_有问题，修改为_test_问题解决）

测试结果：

```
# 测试代码
ccpm_train_split = []
for i in range(0, len(ccpm_train["train"])):
    for j in range(0, 4):
        temp_json = {"translation": ccpm_train["train"][i]["translation"]}
        temp_json["choices"] = ccpm_train["train"][i]["choices"][j]
        if j == int(ccpm_train["train"][i]["answer"]):
            temp_json["answer"] = 1
        else:
            temp_json["answer"] = 0
        ccpm_train_split.append(temp_json)

    if i == 5:
        break

for i in range(0, len(ccpm_train_split)):
    print(ccpm_train_split[i])
```

```
{'translation': '诗人啊，你竟像在遥远的地方站立船头。', 'choices': '行人初上木兰舟', 'answer': 0}
{'translation': '诗人啊，你竟像在遥远的地方站立船头。', 'choices': '骚人遥驻木兰舟', 'answer': 1}
{'translation': '诗人啊，你竟像在遥远的地方站立船头。', 'choices': '有人独上木兰舟', 'answer': 0}
{'translation': '诗人啊，你竟像在遥远的地方站立船头。', 'choices': '行人迢递木兰舟', 'answer': 0}
{'translation': '他的双眼眼瞳碧绿而有光，头发金黄而弯曲，两鬓呈红色。', 'choices': '绿玉觜攒鸡脑破，玄金爪擘兔心开。', 'answer': 0}
{'translation': '他的双眼眼瞳碧绿而有光，头发金黄而弯曲，两鬓呈红色。', 'choices': '翅金肉白顶红麻，项矜毛青腿少瑕。', 'answer': 0}
{'translation': '他的双眼眼瞳碧绿而有光，头发金黄而弯曲，两鬓呈红色。', 'choices': '头似珊瑚项班红，翅如金箔肉带黄。', 'answer': 0}
{'translation': '他的双眼眼瞳碧绿而有光，头发金黄而弯曲，两鬓呈红色。', 'choices': '碧玉炅炅双目瞳，黄金拳拳两鬓红。', 'answer': 1}
```

✓ **预处理**

在进行下一阶段的预处理前，有必要首先对本任务的建模思路进行首先设计，可供采取的思路有：

- 建模思路

思路一：将给定的翻译与每个候选诗行(带有一个额外的分隔符)连接起来，并将连接输入 BERT。然后它将[CLS]标记的隐藏状态提供给线性层，并进行二进制分类：匹配或不匹配。(该方法称为 BERT-Cls，在测试集上的得分为 84.96)

思路二：使用两个 bert 分别对译文和候选诗行进行编码，得到译文和诗行的嵌入。然后计算翻译嵌入与每一行诗之间的余弦相似度。与译文最相似的候选诗被选为最终答案。(该方法称为 BERT-Match，在测试集上的得分为 82.60)

考虑到思路一的效果较思路二略好，这里采用建模思路一。

按照建模思路一，则预处理的第一部分工作就是将给定的翻译与每个候选诗行(带有一个额外的分隔符，这里采用“=”)连接起来。

实现代码：

```
# 1.建模思路一：将翻译和候选诗行进行拼接
ccpm_train_concat = []
for i in range(0, len(ccpm_train_split)):
    temp_json = {"text": ccpm_train_split[i]["translation"] + "=" + ccpm_train_split[i]["choices"],
                 "label": ccpm_train_split[i]["answer"]}
    ccpm_train_concat.append(temp_json)

ccpm_valid_concat = []
for i in range(0, len(ccpm_valid_split)):
    temp_json = {"text": ccpm_valid_split[i]["translation"] + "=" + ccpm_valid_split[i]["choices"],
                 "label": ccpm_valid_split[i]["answer"]}
    ccpm_valid_concat.append(temp_json)

ccpm_test_concat = []
for i in range(0, len(ccpm_test_split)):
    temp_json = {"text": ccpm_test_split[i]["translation"] + "=" + ccpm_test_split[i]["choices"],
                 "label": ccpm_test_split[i]["answer"]}
    ccpm_test_concat.append(temp_json)
```

测试结果：

```
# 测试代码
# 建模思路一：将翻译和候选诗行进行拼接
ccpm_train_concat = []
for i in range(0, len(ccpm_train_split)):
    temp_json = {"text": ccpm_train_split[i]["translation"] + "=" + ccpm_train_split[i]["choices"],
                 "label": ccpm_train_split[i]["answer"]}
    ccpm_train_concat.append(temp_json)
    if i == 5: break

for i in range(0, len(ccpm_train_concat)):
    print(ccpm_train_concat[i])
```

```
{'text': '诗人啊，你竟像在遥远的地方站立船头。=行人初上木兰舟', 'label': 0}
{'text': '诗人啊，你竟像在遥远的地方站立船头。=骚人遥驻木兰舟', 'label': 1}
{'text': '诗人啊，你竟像在遥远的地方站立船头。=有人独上木兰舟', 'label': 0}
{'text': '诗人啊，你竟像在遥远的地方站立船头。=行人迢递木兰舟', 'label': 0}
{'text': '他的双眼眼瞳碧绿而有光，头发金黄而弯曲，两鬓呈红色。=绿玉觜攒鸡脑破，玄金爪擘兔心开。', 'label': 0}
{'text': '他的双眼眼瞳碧绿而有光，头发金黄而弯曲，两鬓呈红色。=翅金肉白顶红麻，项襂毛青腿少瑕。', 'label': 0}
```

- 其余工作

```
# 2.加载句子标记器
tokenizer = AutoTokenizer.from_pretrained("thu-cbert-character")

# 3.定义预处理函数
# 用于标记和截断序列，使其不超过模型的最大输入长度：text
def preproccess_function(examples):
    return tokenizer(examples["text"], truncation=True)

# 4.对整个数据集进行批量化预处理
# 使用数据集映射函数将预处理函数应用于整个数据集
ccpm_train_tokenized = list(map(preproccess_function, ccpm_train_concat))
ccpm_valid_tokenized = list(map(preproccess_function, ccpm_valid_concat))

ccpm_train_tokenized2 = []
for i in range(0, len(ccpm_train_concat)):
    temp_json = {"input_ids": ccpm_train_tokenized[i]["input_ids"],
                 "label": ccpm_train_concat[i]["label"]}
    ccpm_train_tokenized2.append(temp_json)

ccpm_valid_tokenized2 = []
for i in range(0, len(ccpm_valid_concat)):
    temp_json = {"input_ids": ccpm_valid_tokenized[i]["input_ids"],
                 "label": ccpm_valid_concat[i]["label"]}
    ccpm_valid_tokenized2.append(temp_json)

# 5.文本动态填充到统一长度：最长元素的长度
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

测试结果：

```
In 23
# 预处理1-3:
# 1.加载句子标记器以处理字段：text
tokenizer = AutoTokenizer.from_pretrained("thu-cbert-character")

In 24
# 2.定义预处理函数
# 用于标记和截断序列，使其不超过模型的最大输入长度：text
def preproccess_function(examples):
    return tokenizer(examples["text"], truncation=True)

In _
# 3.对整个数据集进行批量化预处理
# 使用数据集映射函数将预处理函数应用于整个数据集
ccpm_train_tokenized = list(map(preproccess_function, ccpm_train_concat))
ccpm_valid_tokenized = list(map(preproccess_function, ccpm_valid_concat))

In 26
ccpm_train_tokenized

Out 26
[{'input_ids': [2, 849, 101, 980, 77, 190, 1041, 539, 97, 1878, 648, 94, 128, 141, 676, 324, 1118, 342, 76, 60, 127, 101, 694, 115,
882, 932, 2193, 3], 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]},
 {'input_ids': [2, 849, 101, 980, 77, 190, 1041, 539, 97, 1878, 648, 94, 128, 141, 676, 324, 1118, 342, 76, 60, 2652, 101, 1878,
 1619, 882, 932, 2193, 3], 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]},
```

✓ 模型训练与验证

参考超参：

- 句子编码器：Transformer 库中选择 bert-base-chinese / 实验提供的模型。
- 微调：使用 Adam 优化器，初始学习率 2e−5 线性下降，训练模型 4 个 epoch。

```python
# 训练
# 使用AutoModelForSequenceClassification加载模型以及标签的数量
model = AutoModelForSequenceClassification.from_pretrained("thu-cbert-character", num_labels=2)

# 在训练参数中定义训练超参数
training_args = TrainingArguments(
    output_dir="./results",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=4,
    weight_decay=0.01,
    save_total_limit=2,
    remove_unused_columns=False,
)

# 将训练参数与模型、数据集、分词器和数据整理器一起传递给训练器
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=ccpm_train_tokenized2,
    eval_dataset=ccpm_valid_tokenized2,
    tokenizer=tokenizer,
    data_collator=data_collator,
)

trainer.train()
```

测试结果：

```
In 34  1  # 微调模型
       2  trainer.train()

  /home/djn/anaconda3/envs/CLS/lib/python3.10/site-packages/transformers/optimization.py:306: FutureWarning: This implementation of
  AdamW is deprecated and will be removed in a future version. Use the PyTorch implementation torch.optim.AdamW instead, or set
  `no_deprecation_warning=True` to disable this warning
    warnings.warn(
  ***** Running training *****
    Num examples = 97992
    Num Epochs = 1
    Instantaneous batch size per device = 16
    Total train batch size (w. parallel, distributed & accumulation) = 16
    Gradient Accumulation steps = 1
    Total optimization steps = 6125
    Number of trainable parameters = 108031490
  You're using a BertTokenizerFast tokenizer. Please note that with a fast tokenizer, using the `__call__` method is faster than using
   a method to encode the text followed by a call to the `pad` method to get a padded encoding.

  Step   Training Loss
```

```
Configuration saved in ./results/checkpoint-5000/config.json
Model weights saved in ./results/checkpoint-5000/pytorch_model.bin
tokenizer config file saved in ./results/checkpoint-5000/tokenizer_config.json
Special tokens file saved in ./results/checkpoint-5000/special_tokens_map.json
Saving model checkpoint to ./results/checkpoint-5500
Configuration saved in ./results/checkpoint-5500/config.json
Model weights saved in ./results/checkpoint-5500/pytorch_model.bin
tokenizer config file saved in ./results/checkpoint-5500/tokenizer_config.json
Special tokens file saved in ./results/checkpoint-5500/special_tokens_map.json
Saving model checkpoint to ./results/checkpoint-6000
Configuration saved in ./results/checkpoint-6000/config.json
Model weights saved in ./results/checkpoint-6000/pytorch_model.bin
tokenizer config file saved in ./results/checkpoint-6000/tokenizer_config.json
Special tokens file saved in ./results/checkpoint-6000/special_tokens_map.json
```

```
In 30  1 trainer.evaluate()

       ***** Running Evaluation *****
         Num examples = 10880
         Batch size = 16
       [1/680 : < :]

Out 30 {'eval_loss': 0.5118282437324524,
        'eval_runtime': 4.7525,
        'eval_samples_per_second': 2289.301,
        'eval_steps_per_second': 143.081,
        'epoch': 1.0}
```

✓ 模型测试

```
# 模型测试
ccpm_test_tokenized = list(map(preproccess_function, ccpm_test_concat))

ccpm_test_tokenized2 = []
for i in range(0, len(ccpm_test_concat)):
    temp_json = {"text": ccpm_test_concat[i]["text"],
                 "input_ids": ccpm_test_tokenized[i]["input_ids"],
                 "attention_mask":ccpm_test_tokenized[i]["attention_mask"],
                 }
    ccpm_test_tokenized2.append(temp_json)

test_predictions = trainer.predict(ccpm_test_tokenized2)
test_predictions_argmax = np.argmax(test_predictions[0], axis=1)
test_references = np.array()
for i in range(0, len(ccpm_test_tokenized2)):
    test_references.append(ccpm_test_tokenized2[i]["label"])
metric.compute(predictions=test_predictions_argmax, references=test_references)
```

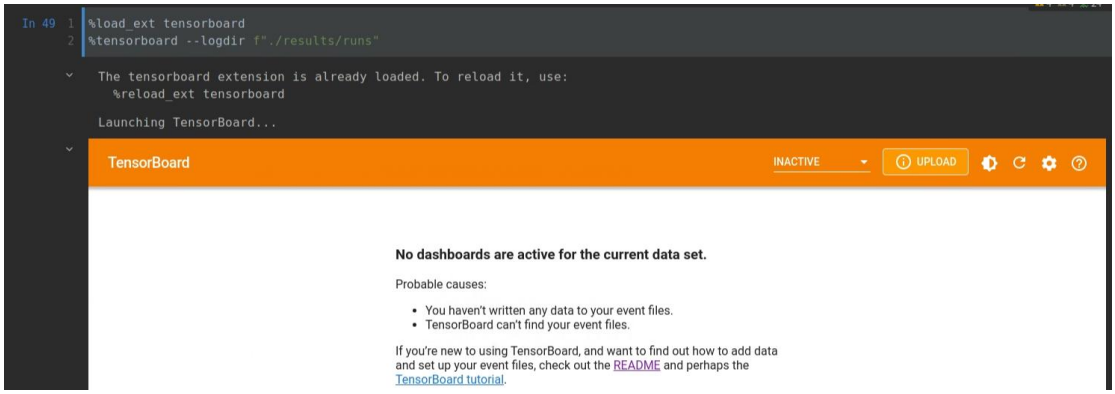
测试结果：

```
In 21  1 # 模型测试
       2 ccpm_test_tokenized = list(map(preproccess_function, ccpm_test_concat))
       3
       4 ccpm_test_tokenized2 = []
       5 for i in range(0, len(ccpm_test_concat)):
       6     temp_json = {"text": ccpm_test_concat[i]["text"],
       7                  "input_ids": ccpm_test_tokenized[i]["input_ids"],
       8                  "attention_mask":ccpm_test_tokenized[i]["attention_mask"],
       9                  }
      10     ccpm_test_tokenized2.append(temp_json)

In 26  1 print(ccpm_test_tokenized2[0])

       {'text': '侍奉夫主，不能尽自己的天年。=事主不尽年', 'input_ids': [101, 1765, 100, 1813, 1747, 1989, 1744, 100, 100, 100, 100, 1916, 1811,
       1840, 1636, 1027, 1751, 1747, 1744, 100, 1840, 102], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1]}
```

2. 超参调优

Step1：利用验证集（valid.jsonl）进行checkpoint 选择和超参调整

为方便超参的调优，保存训练过程中的 log，并采用 TensorBoard 绘制训练 loss 曲线。



并据此依次修改上述相关代码：如新增计算准确率函数和 log 相关超参

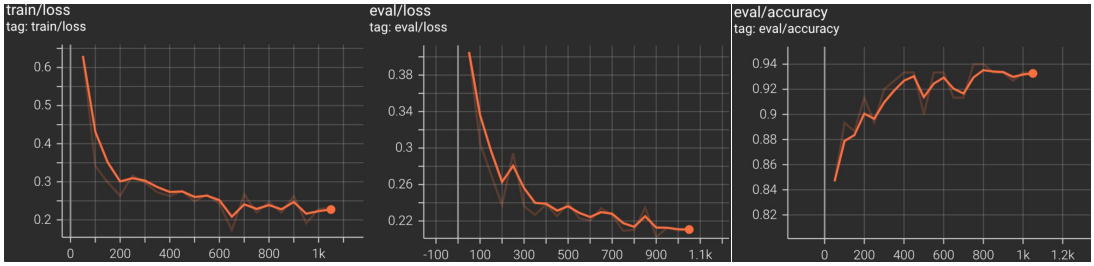
```python
metric = load_metric("accuracy")
def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    predictions = np.argmax(predictions, axis=1)
    results = metric.compute(predictions=predictions, references=labels)
    return results

# 在训练参数中定义训练超参数
training_args = TrainingArguments(
    output_dir="./results",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=1,
    weight_decay=0.01,

    evaluation_strategy="steps",
    eval_steps=50,
    logging_strategy="steps",
    logging_steps=50,
    save_strategy="steps",
    save_steps=1000,
    load_best_model_at_end=True,
    metric_for_best_model="accuracy",
```

Step2：超参调优

✓ 本项目涉及到的主要超级参数包括

参数名称	参数用途	最优取值
learning_rate	学习率	2e−5
batch_size	批大小	16（for train and for eval）
epochs	批次	4

✓ 在训练过程中，刷新 TensorBoard 来查看训练指标的更新，包括训练集上的损失、验证集上的损失和验证集上的准确率。



3. 模型测试与样例分析

Step1：最终在测试集（test_public.jsonl）上进行预测

与智源指数 CUGE 评测上其他模型的预测结果进行对比

排行	模型名	Accuracy
1	CPM-Ant	92.13
2	MultiPrompt	91.91
3	CPM-2	91.60
4	mT5-XXL	90.60
5	bert-MULTIPR OMPT	90.07
Our	thu-cbert-character	< 80

✓ 受限于本地可用空间不足，以及显卡可用内存有限，没有进行多轮参数调优，所以结果仍有较大的可提升空间：

✓ 下一步工作：

1. 使用 Trainer 调整超参数。
2. 微调其他更好的模型。
3. 使用其他方式进行训练。

Step2：case study

模型在测试集上错例和正例的采样分析

✓ 错例分析

translation	choices	answer	predict
柴烟中红星乱闪。	红光生紫烟	0	1
	红星乱紫烟	1	0

由于两句预选诗句同位置相同字占比很高，且其中错误预选诗句和翻译中相同字占比很高，所以在模型不能较好学习语义时，会受到这样的雨轩诗句误导，做出错误的判断。

✓ 正例分析

translation	choices	answer	predict
清晨还是西北风	清晨西北转	1	1
	河岳西来转	0	0
	凌晨从北固	0	0
	西北转银潢	0	0

由于四句预选诗句同位置相同字占比并不高，且其中错误预选诗句和翻译中相同字占比也不高，反而正确预选诗句和翻译中相同字占比非常高，所以在模型科可以相对轻松地做出正确判断。

二、语言理解能力-词句级 | 命名实体识别

从以下三个改进方向中选择一项完成：

- 尝试其他的中文预训练模型（可以在 transformers 社区上查找，分析模型区别，优劣。【进行中】
- 尝试智源指数 CUGE 其他分类数据集，进行打榜。【选择】
- 尝试使用不同的方法解决 CCPM 任务，分析各方法优劣，性能。【在上一章节的代码中，同时对比了其他 bert 模型】

1. 代码编写

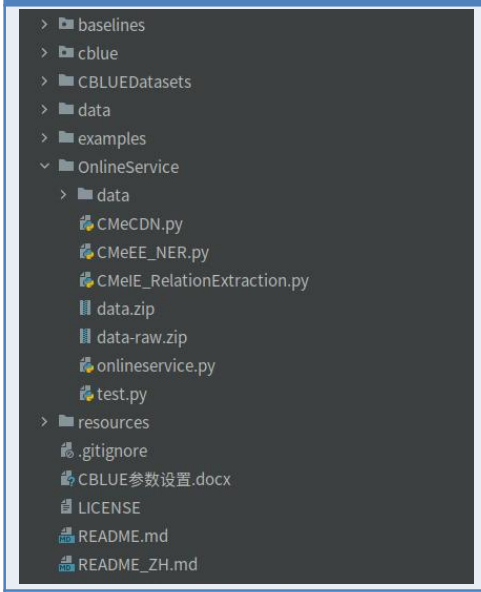
Step1：下载数据集 CCPM

- ✓ CMeEE 数据集主要用于医学实体识别任务。
- ✓ 该任务共标注了 938 个文件，47,194 个句子
- ✓ 包含了 504 种常见的儿科疾病、7,085 种身体部位、12,907 种临床表现、4,354 种医疗程序等 9 大类医学实体
- ✓ 将医学文本命名实体划分为九大类，包括：疾病(dis)，临床表现(sym)，药物(dru)，医疗设备(equ)，医疗程序(pro)，身体(bod)，医学检验项目(ite)，微生物类(mic)，科室(dep)
- ✓ 标注之前对文章进行自动分词处理，所有的医学实体均已正确切分
- ✓ 训练、验证和测试集分别为 15,000、5,000 和 3,000 条


```
{
  "text": "（5）房室结消融和起搏器植入作为反复发作或难治性心房内折返性心动过速的替代疗法。",
  "entities": [
    {
      "start_idx": 3,
      "end_idx": 7,
      "type": "pro",
      "entity": "房室结消融"
    },
    {
      "start_idx": 9,
      "end_idx": 13,
      "type": "pro",
      "entity": "起搏器植入"
    },
    {
      "start_idx": 16,
      "end_idx": 33,
      "type": "dis",
      "entity": "反复发作或难治性心房内折返性心动过速"
    }
  ]
}
```


Step2：构建 Python 项目

构建后项目的整体结构如下：

Pycharm 截图	一级目录	二级目录	文件用途
	CMeEE	baselines	实现基线模型，模型来源于网络公开
		cblue CBLUEDatasets	引入外部数据集，丰富本模型的训练
		data	本项目需要的源数据
		examples	预训练模型参数
		OnlineService	以 python-app 形式最后将命名实体识别进行了接口封装
		resources	引入的其他资源：包括医疗数据库等

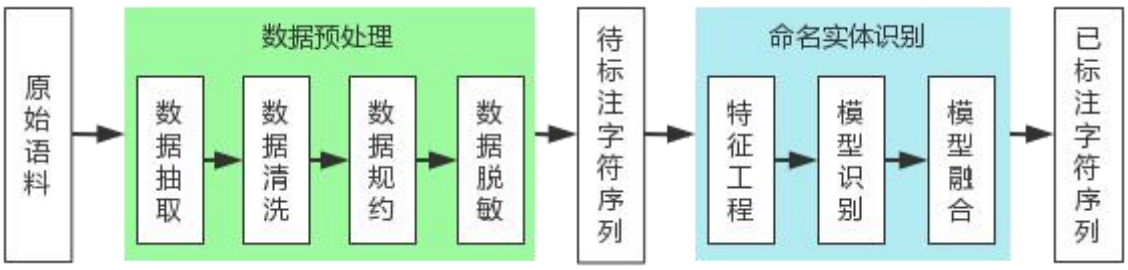
项目环境配置为：CUDA11.6　Python3.10
需要额外依赖包：较多，这里略去

Step3：书写核心代码

✓ 处理流程

医学实体识别的一般流程包括：

- 先将原始语料进行数据抽取、清洗、规约与脱敏四步预处理，获得待标记的字符序列
- 之后将其输入命名实体识别模型中进行计算，获得标注好的字符序列作为最终结果。
- 具体到命名实体识别模型，通常由特征工程、识别方法所对应的模型识别和模型融合三部分构成。



✓ 方法选择

医学实体识别模型的研究主要有三种方法。它们分别是基于词典的方法、基于规则的方法和基于机器学习的方法。方法各有优劣。

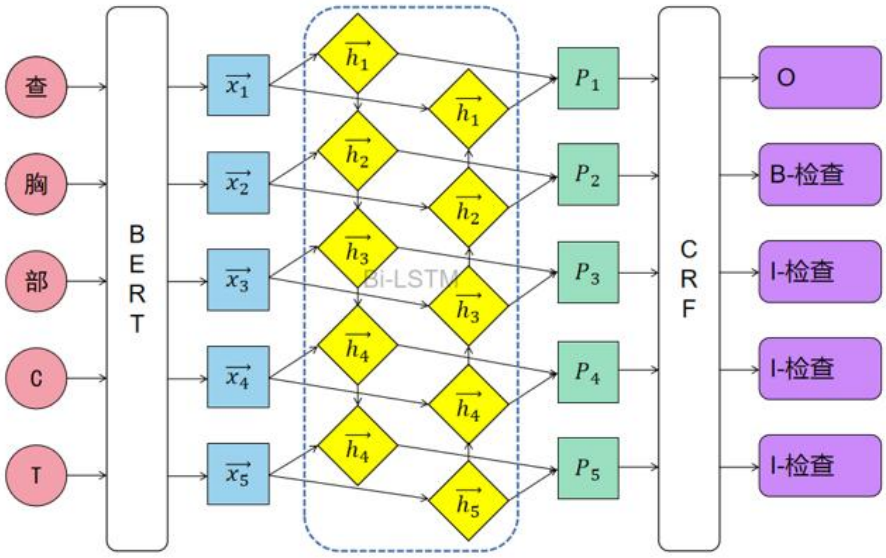
一级分类	二级分类	优点		缺点	模型示例
基于词典		实现简单		• 词典的规模和质量对识别结果有重要影响 • 难以保证及时更新对新增或者补充实体信息的兼容和覆盖	DLAM
基于规则		便于维护		• 需要大量人力和时间成本投入 • 规则的可移植性较差	Rule-based
基于机器学习	统计机器学习	• 实用 • 可移植	-	• 需要有大规模高质量的标注数据集作为训练原料	CRF
	深度学习		准确		BERT

✓ 模型设计

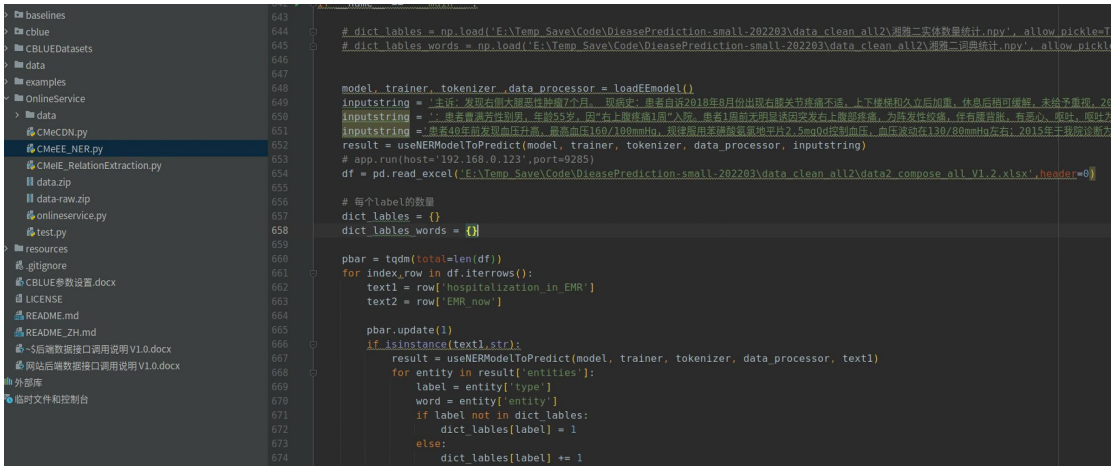
BERT-BiLSTM-CRF(BBC)模型为近年来最受欢迎且效果相对较好的主流模型组合，故采用之。

BBC 模型具体由三部分组成：

- BERT 预训练模型用于解决从文本中提取特征时存在的特征稀疏问题，充分获取医学文本的字粒度信息
- Bi-LSTM 在 BERT 基础上，双向利用上下文信息对句子进行建模
- CRF 考虑标签间的依赖关系，为 Bi-LSTM 预测的标签添加约束以保证最终预测标签的合法性，进一步提升模型准确性



✓ 代码实现



（CMeEE 核心代码共计 707 行，在舱口中进行了整理，由于 baseline 中其他基线模型实现未获得对方版权允许，故相关内容未整理到最终提交的代码中）

2. 实现结果

Step1：测评指标

指标有三，分别是准确率(Precision,简记为 *Prec*)、召回率(Recall,简记为 *Rec*)和 F_1-Measure(简记为 F_1)值。

✓ 准确率衡量命名实体识别模型识别出的所有实体中有多少为正确识别

✓ 召回率衡量命名实体识别模型在整个语料库中正确识别实体的数量占比

✓ F_1 取两者的调和平均值。

设模型正确识别的相关实体数为 T_p，模型错误识别的不相关实体数为 F_p，模型未识别的相关实体数为 F_N。则

$$Prec = \frac{T_p}{T_p + F_p} \times 100\%$$

$$Rec = \frac{T_p}{T_p + F_N} \times 100\%$$

$$F_1 = \frac{2 \times Prec \times Rec}{Prec + Rec} \times 100\%$$

Step2：超参选择

✓ 涉及到的超参包括：

- max_length 表示 BERT 模型输入的最长文本序列长度
- hidden_dim 表示 LSTM 隐层单元/节点个数（这里指单方向的隐层节点数）
- dropout_rate 表示在训练过程中的 dropout 概率,在 BiLSTM 输出部分使用 Dropou 技术减少过拟合
- optimizer 表示模型所采用的优化算法
- epoch 表示迭代次数
- batch_size 表示数据分批后逐批次大小
- learning_rate 表示模型的学习率

✓ 可选方案：

参数/方案序号	1	2	3	4
max_length	200	128	200	128
hidden_dim	128	256	128	64
dropout_rate	0.5	0.5	0.5	0.5
optimizer	Adam	Adam	AdamW	AdamW
epoch	5	10	10	5
batch_size	16	8	8	16
learning_rate	5e-5	5e-5	5e-5	5e-5
最终 F_1	60.02	62.08	63.74	65.58

Step3：实现结果

模型	Prec	Rec	F_1
BERT-BiLSTM-CRF	64.49	62.64	65.58

三、附录

1. 代码仓库

本次作业的全部代码在 github 上进行了开源，相关代码仍在继续整理中，具体访问地址如下：

- 远程仓库的名称　greengrasscugb/CL_NLP_CCPM-CMeEE
- 远程仓库的地址 https://github.com/greengrasscugb/CL_NLP_CCPM-CMeEE.git

2. 参考资料

[1] 助教提供代码下载 https://cloud.tsinghua.edu.cn/d/906d7692ef6a4098b225/
[2] 数据集下载，智源指数 (baai.ac.cn)，
http://cuge.baai.ac.cn/#/dataset?id=1&name=CCPM
[3] 使用 HuggingFace 微调 BERT：
https://cloud.tencent.com/developer/article/1980898
[4] Hugging Face 文本分类：文本分类是一项常见的 NLP 任务，用于为文本分配标签或类。
https://huggingface.co/docs/transformers/v4.24.0/en/tasks/sequence_classification#load-imdb-dataset
[5] BERT 实战（1）：使用 DistilBERT 作为词嵌入进行文本情感分类，与其它词向量对比（FastText,Word2vec,Glove）
https://blog.csdn.net/qq_39610915/article/details/118556253
[6] Pytorch 使用 BERT 预训练模型微调文本分类，IMDb 电影评论数据集
https://blog.csdn.net/Code_Tookie/article/details/104944888