

研究生《自然语言处理》课程

实验报告

|  |  |
| --- | --- |
| **实验名称：** | **命名实体识别** |
| **组 号：** | **11** |
| **组 长：** | **24140617\_王昭** |
| **组 员：** | **24140618\_王子豪** |
| **组 员：** | **24140580\_贾登辉** |
| **组 员：** | **24140625\_杨昌霖** |
| **日 期：** | **2024.11.16** |

## **（1）实验目的**

1. 了解命名实体识别任务，以及它的应用场景；

2. 综合考虑各种算法的优劣势，并选择适合中文命名实体识别的算法； 3. 在给定数据集上完成中文命名实体识别任务。

## **（2）实验环境**

硬件环境:

GPU：NVIDIA RTX 4090

显存：24G

软件环境:

·IDE：PyCharm Professional

·主要依赖库: scikit-learn==1.1.3

scipy==1.10.1

seqeval==1.2.2

transformers==4.27.4

pytorch-crf==0.7.2

## **（3）实验原理**

### 本实验的模型核心模块由BERT+BiLSTM+CRF组成，接下来我将对每一部分的原理和功能进行具体分析

**BERT工作原理：**

1. BERT是基于Transformer的预训练语言模型，能够提供子词级的上下文表示（即一个词的向量表示会根据上下文动态变化）。
2. NER任务中，BERT用作特征提取器，为每个词生成高质量的嵌入表示。

**BiLSTM工作原理：**

1. LSTM是一种特殊的RNN，擅长处理序列数据并保留长期依赖信息。
2. 双向LSTM（BiLSTM）能够同时考虑前向和后向的上下文信息，从而生成更加全面的序列表示。

**CRF:**

1. 条件随机场（CRF）是一种概率模型，用于对序列数据的标注进行全局优化。
2. 它利用标签之间的依赖关系，通过最大化路径的条件概率来确保生成合法的标签序列。

## **（4）实验内容**

### **4.1数据处理**

该模块的总目标：将原始文本和对应的命名实体标签（NER标签）转化为适合模型输入和训练的数据格式，同时确保特征和标签的对齐。具体目标包括以下几个方面：

**数据读取与BIO标注：**

1. def convert\_to\_bio\_format\_txt(self, \_data, \_file\_name, max\_seq\_len=128):
2. """
3. 将输入数据转换为 BIO 格式，并保存到本地
4. :param self: 自身对象
5. :param \_data: ori\_data 格式的原始数据
6. :param \_file\_name: str, 保存的文件名
7. :param max\_seq\_len: int, 句子的最大长度, 默认为 128
8. :return: None
9. """
10. \_txt\_data = []
11. \_replace\_en\_sign = 0
12. for \_item in \_data:
13. \_sentence = \_item.get('sentence', '')
14. \_entities = \_item.get('entities', [])
15. *# 繁体转简体*
16. converter = opencc.OpenCC('t2s')  *# 繁体转简体*
17. \_sentence = converter.convert(\_sentence)
18. *# 英文部分字符转换为中文字符*
19. punctuation\_map = {
20. ',': '，',
21. ':': '：',
22. ';': '；',
23. '!': '！',
24. '?': '？',
25. '(': '（',
26. ')': '）',
27. '[': '【',
28. ']': '】'
29. }
30. *# 替换文本中的英文标点符号为中文标点符号*
31. for en\_punct, zh\_punct in punctuation\_map.items():
32. \_sentence = \_sentence.replace(en\_punct, zh\_punct)
33. \_replace\_en\_sign += 1
34. *# logger.info(f"Replace English punctuation to Chinese punctuation: {\_replace\_en\_sign}")*
35. if len(\_sentence) > max\_seq\_len:
36. *# 分割句子*
37. merged\_sentences, start\_indices = cut\_sent(\_sentence, max\_seq\_len)
38. *# 获得句子长度列表*
39. sent\_len\_list = [len(sent) for sent in merged\_sentences]
40. sent\_index = 0
41. all\_entity = []
42. cur\_entity = []
43. for \_entity in \_entities:
44. \_entity\_pos = \_entity['pos']
45. *# 遍历实体，获取实体的开始和结束位置*
46. entity\_start = \_entity\_pos[0]
47. entity\_end = \_entity\_pos[1]
48. *# 获取句子的开始和结束位置*
49. sent\_start = start\_indices[sent\_index]
50. sent\_end = start\_indices[sent\_index] + len(merged\_sentences[sent\_index])
51. *# 如果实体的开始和结束位置在句子的开始和结束位置之间，加入到当前句子的实体列表中*
52. if entity\_start >= sent\_start and entity\_end <= sent\_end:
53. cur\_entity.append(\_entity)
54. *# 如果实体的开始位置大于了句子的结束位置之间，将当前句子的实体列表加入到所有实体列表中，*
55. *# 重置当前句子的实体列表，更新句子索引，加入当前实体*
56. else:
57. all\_entity.append(cur\_entity)
58. cur\_entity = []
59. sent\_index += 1
60. cur\_entity.append(\_entity)
61. *# 将最后一个句子的实体列表加入到所有实体列表中*
62. all\_entity.append(cur\_entity)
63. *# 累加的句子长度*
64. \_sum\_sent\_len = 0
65. for index, \_entity in enumerate(all\_entity):
66. *# 从下标为 1 开始，累加句子长度*
67. if index > 0:
68. \_sum\_sent\_len += sent\_len\_list[index - 1]
69. for \_index, item in enumerate(\_entity):
70. *# 从下标为 1 开始，调整实体的位置*
71. item['pos'][0] -= \_sum\_sent\_len
72. item['pos'][1] -= \_sum\_sent\_len
73. linked\_sent\_entities = []
74. for sent, \_entity in zip(merged\_sentences, all\_entity):
75. linked\_sent\_entities.append({'sentence': sent, 'entities': \_entity})
76. for linked\_sent\_entity in linked\_sent\_entities:
77. \_sent = linked\_sent\_entity.get('sentence', '')
78. \_sent = [i for i in \_sent]
79. \_entities = linked\_sent\_entity.get('entities', [])
80. *# print(\_entities)*
81. \_labels = ["O"] \* len(\_sent)
82. for \_entity in \_entities:
83. start, end = \_entity["pos"]
84. entity\_type = \_entity["type"]
85. \_labels[start] = f"B-{entity\_type}"
86. for i in range(start + 1, end):
87. \_labels[i] = f"I-{entity\_type}"
88. \_each\_data = {
89. "text": \_sent,
90. "labels": \_labels
91. }
92. \_txt\_data.append(\_each\_data)
93. else:
94. \_sent = [i for i in \_sentence]
95. \_labels = ["O"] \* len(\_sent)
96. for \_entity in \_entities:
97. start, end = \_entity["pos"]
98. entity\_type = \_entity["type"]
99. \_labels[start] = f"B-{entity\_type}"
100. for i in range(start + 1, end):
101. \_labels[i] = f"I-{entity\_type}"
102. \_each\_data = {
103. "text": \_sent,
104. "labels": \_labels
105. }
106. \_txt\_data.append(\_each\_data)
107. \_pre\_path = os.path.join(os.path.dirname(self.file\_path), 'pre\_data')
108. *# 如果文件路径不存在，则创建路径*
109. if not os.path.exists(\_pre\_path):
110. os.makedirs(\_pre\_path)
111. logger.info(f"Create directory: {\_pre\_path}")
112. else:
113. logger.info(f"Directory exists: {\_pre\_path}")
114. *# 保存到本地*
115. with open(os.path.join(\_pre\_path, \_file\_name), 'w', encoding='utf-8') as train\_f:
116. logger.info(f"Save data to: {os.path.join(self.file\_path, \_file\_name)}")
117. train\_f.write("\n".join([json.dumps(d, ensure\_ascii=False) for d in \_txt\_data]))
118. logger.info(f"Save complete! Save txt \"{\_file\_name}\", size: {len(\_txt\_data)}")

**处理单字符断句符、英文省略号和中文省略号：**

1. def cut\_sentences\_v1(sent):

    sent = re.sub('([。！？\?])([^”’])', r"\1\n\2", sent)  *# 单字符断句符*

1. sent = re.sub('(\.{6})([^”’])', r"\1\n\2", sent)  *# 英文省略号*
2. sent = re.sub('(\…{2})([^”’])', r"\1\n\2", sent)  *# 中文省略号*
3. sent = re.sub('([。！？\?][”’])([^，。！？\?])', r"\1\n\2", sent)  *# 双引号后的处理*
4. *# 如果双引号前有终止符，那么双引号才是句子的终点，把分句符\n放到双引号后*
5. return sent.split("\n")  *# 用换行符切分成句子列表*

**切分文本为句子，合并句子并确保每个句子长度不超过最大限制：**

1. def cut\_sent(text, max\_seq\_len):
2. """
3. 切分文本为句子，合并句子并确保每个句子长度不超过最大限制
4. :param text: 原始文本
5. :param max\_seq\_len: 句子的最大长度
6. :return: 合并后的句子列表和每个句子的起始下标
7. """
8. sentences = []  *# 存储所有切分的句子*
9. start\_indices = []  *# 存储每个句子的起始下标*
10. *# 细粒度划分*
11. sentences\_v1 = cut\_sentences\_v1(text)
12. current\_index = 0  *# 当前字符索引*
13. for sent\_v1 in sentences\_v1:
14. if len(sent\_v1) > max\_seq\_len - 2:
15. *# 如果句子超长，进行二次切分*
16. sentences\_v2 = cut\_sentences\_v2(sent\_v1)
17. for s in sentences\_v2:
18. sentences.append(s)
19. start\_indices.append(current\_index)  *# 记录起始位置*
20. current\_index += len(s)  *# 更新当前索引*
21. else:
22. sentences.append(sent\_v1)
23. start\_indices.append(current\_index)  *# 记录起始位置*
24. current\_index += len(sent\_v1)  *# 更新当前索引*
25. assert ''.join(sentences) == text  *# 确保切分后能重新拼接成原文本*
26. *# 合并句子*
27. merged\_sentences = []  *# 存储合并后的句子*
28. merged\_start\_indices = []  *# 存储合并句子的起始下标*
29. start\_index\_ = 0  *# 当前合并句子的起始索引*
30. while start\_index\_ < len(sentences):
31. tmp\_text = sentences[start\_index\_]  *# 当前合并的句子*
32. tmp\_start\_index = start\_indices[start\_index\_]  *# 当前句子的起始下标*
33. end\_index\_ = start\_index\_ + 1  *# 合并的下一个句子索引*
34. *# 根据最大长度合并句子*
35. while end\_index\_ < len(sentences) and \
36. len(tmp\_text) + len(sentences[end\_index\_]) <= max\_seq\_len - 2:
37. tmp\_text += sentences[end\_index\_]  *# 添加到当前句子*
38. end\_index\_ += 1
39. start\_index\_ = end\_index\_  *# 更新合并的开始索引*
40. merged\_sentences.append(tmp\_text)  *# 添加合并后的句子*
41. merged\_start\_indices.append(tmp\_start\_index)  *# 添加起始下标*
42. return merged\_sentences, merged\_start\_indices  *# 返回合并后的句子和起始下标*

**为了更好的测试效果将 train\_data 和 dev\_data 组合到一起，并按照比例重新划分：**

1. def combine\_and\_split(\_train\_data, \_dev\_data, train\_ratio=0.95):
2. *# 将 train\_data 和 dev\_data 合并*
3. combined\_data = \_train\_data + \_dev\_data
4. logger.info(f"Combined data size: {len(combined\_data)}")
5. *# 使用 sklearn 的 train\_test\_split 按比例划分*
6. new\_train\_data, new\_dev\_data = train\_test\_split(combined\_data, test\_size=1 - train\_ratio, random\_state=42)
7. logger.info(f"New train data size: {len(new\_train\_data)}")
8. logger.info(f"New dev data size: {len(new\_dev\_data)}")
9. return new\_train\_data, new\_dev\_data

### **4.2模型构建**

**设计模型三大核心组件：BERT+BiLSTM+CRF**

**BERT预训练模型**：将用中文预训练好的BERT模型，从Hugging Face下载到本地并加载

1. self.bert = AutoModel.from\_pretrained("weiweishi/roc-bert-base-zh")
2. self.bert\_config = AutoConfig.from\_pretrained("weiweishi/roc-bert-base-zh")

**初始化BiLSTM模型**：

1. *# BiLSTM 隐藏层大小*
2. if args.model\_name == "bert\_wwn\_ext\_base":
3. self.lstm\_hiden = 512
4. else:
5. self.lstm\_hiden = 256
6. *# 最大序列长度*
7. self.max\_seq\_len = args.max\_seq\_len
8. *# BiLSTM 层, 输入大小为 hidden\_size，输出大小为 2\* self.lstm\_hiden*
9. self.bilstm = nn.LSTM(hidden\_size, self.lstm\_hiden, 1, bidirectional=True, batch\_first=True)
10. self.dropout = nn.Dropout(0.1)
11. *# 线性层，将 BiLSTM 输出转换为标签数量大小*
12. self.linear = nn.Linear(self.lstm\_hiden \* 2, args.num\_labels)
13. *# CRF 层，用于优化序列预测，batch\_first=True 表示输入的 batch\_size 在第一维*
14. self.crf = CRF(args.num\_labels, batch\_first=True)

**CRF**：内嵌在前向传播的过程中，对前两个模型联合处理结果得到的序列数据进行标注，保证各数据之间的合法性：

1. def forward(self, input\_ids, attention\_mask, labels=None):
2. *# Bert 输入为 input\_ids 和 attention\_mask*
3. bert\_output = self.bert(input\_ids=input\_ids, attention\_mask=attention\_mask)
4. *# 输出 bert\_output[0]，即最后一层隐藏状态。*
5. seq\_out = bert\_output[0]  *# [batchsize, max\_len, 768]*
6. batch\_size = seq\_out.size(0)
7. *# 将上一层（BERT）的输出 seq\_out 传入 BiLSTM 层*
8. *# seq\_out：形状为 [batch\_size, max\_seq\_len, lstm\_hidden \* 2]，代表每个时间步的隐藏状态（双向，因此是 lstm\_hidden \* 2）*
9. *# \_：LSTM的最后一个时间步的隐藏状态，这里没有用到*
10. seq\_out, \_ = self.bilstm(seq\_out)
11. seq\_out = self.dropout(seq\_out)
12. *# 将张量展平，为了将数据展平，以适应全连接层的输入要求*
13. seq\_out = seq\_out.contiguous().view(-1, self.lstm\_hiden \* 2)
14. *# 在全连接层处理后，将数据重塑回原始的序列形状，以便 CRF 层可以正确处理序列数据*
15. seq\_out = seq\_out.contiguous().view(batch\_size, self.max\_seq\_len, -1)
16. *# 线性层处理 BiLSTM 输出，将 LSTM 的输出转换为适合 CRF 层的形状*
17. seq\_out = self.linear(seq\_out)
18. *# 使用CRF解码预测的序列标签（logits），使用 attention\_mask 来忽略无效的位置*
19. logits = self.crf.decode(seq\_out, mask=attention\_mask.bool())
20. loss = None
21. *# 如果提供了真实标签 labels，计算负对数似然损失 loss*
22. if labels is not None:
23. loss = -self.crf(seq\_out, labels, mask=attention\_mask.bool(), reduction='mean')
24. *# 将预测结果、真实标签和损失包装成 ModelOutput*
25. model\_output = ModelOutput(logits, labels, loss)
26. return model\_output

### **4.3超参数配置**

为了实现更快捷且精确的训练效果，在模型训练过程中添加了**学习率自动调优**的逻辑，根据损失函数的变化自动的调整学习率的大小。同时为了防止过拟合，训练中添加了**正则化**（weight\_decay）。这部分代码统一写在了optimizer\_grouped\_parameters函数中：

1. *# 构建优化器参数组*
2. optimizer\_grouped\_parameters = [
3. *# BERT模块参数组，不包含不衰减的参数*
4. {"params": [p for n, p in bert\_param\_optimizer if not any(nd in n for nd in no\_decay)],
5. "weight\_decay": args.weight\_decay, 'lr': args.bert\_learning\_rate},
6. *# BERT 模块参数，包含不衰减的参数*
7. {"params": [p for n, p in bert\_param\_optimizer if any(nd in n for nd in no\_decay)],
8. "weight\_decay": 0.0, 'lr': args.bert\_learning\_rate},
9. *# 其他模块，差分学习率*
10. {"params": [p for n, p in other\_param\_optimizer if not any(nd in n for nd in no\_decay)],
11. "weight\_decay": args.weight\_decay, 'lr': args.crf\_learning\_rate},
12. *# 其他模块，不包含不衰减的参数*
13. {"params": [p for n, p in other\_param\_optimizer if any(nd in n for nd in no\_decay)],
14. "weight\_decay": 0.0, 'lr': args.crf\_learning\_rate},
15. ]
16. *# 使用 AdamW 优化器*
17. optimizer = AdamW(optimizer\_grouped\_parameters, lr=args.bert\_learning\_rate, eps=args.adam\_epsilon)
18. *# 使用线性预热调度器*
19. scheduler = get\_linear\_schedule\_with\_warmup(
20. optimizer, num\_warmup\_steps=int(args.warmup\_proportion \* t\_total), num\_training\_steps=t\_total
21. )
22. return optimizer, scheduler

### **4.4模型训练**

模型训练则是标准流程，数据准备->模型、优化器、损失函数初始化->周期训练。同样为了防止过拟合，在训练过程中引入patience参数进行早停处理（若在patience周期内，验证过程中损失函数没变化则停止训练）。代码如下:

1. main(\_data\_name, \_model\_name):
2. """
3. 主函数，用于训练NER模型
4. :param \_data\_name: 数据集名称
5. :param \_model\_name: BERT 名称
6. :return: None
7. """
8. *# 读取配置参数*
9. args = NerConfig(\_data\_name, \_model\_name)
10. *# 加载 BERT 分词器*
11. logger.info(f"Loading BERT tokenizer from {args.bert\_dir}")
12. if \_model\_name == "roc\_bert":
13. tokenizer = RoCBertTokenizer.from\_pretrained("weiweishi/roc-bert-base-zh")
14. else:
15. tokenizer = BertTokenizer.from\_pretrained(args.bert\_dir)
16. *# 保存模型参数配置*
17. with open(os.path.join(args.output\_dir, \_data\_name + "\_args.json"), "w", encoding='utf-8') as fp:
18. json.dump(vars(args), fp, ensure\_ascii=False, indent=2)
19. *# 读取训练数据*
20. with open(os.path.join(args.data\_path, "train.txt"), "r", encoding='utf-8') as fp:
21. train\_data = fp.read().split("\n")
22. train\_data = [json.loads(d) for d in train\_data]
23. *# 读取验证数据*
24. with open(os.path.join(args.data\_path, "dev.txt"), "r", encoding='utf-8') as fp:
25. dev\_data = fp.read().split("\n")
26. dev\_data = [json.loads(d) for d in dev\_data]
27. *# 创建训练数据集和验证数据集*
28. train\_dataset = NerDataset(train\_data, args, tokenizer)
29. dev\_dataset = NerDataset(dev\_data, args, tokenizer)
30. *# 创建训练数据加载器和验证数据加载器*
31. train\_loader = DataLoader(train\_dataset, shuffle=True, batch\_size=args.train\_batch\_size, num\_workers=2)
32. dev\_loader = DataLoader(dev\_dataset, shuffle=False, batch\_size=args.dev\_batch\_size, num\_workers=2)
33. *# 初始化 NER 模型，并将模型加载到设备上*
34. model = BertNer(args)
35. *# 检查CUDA是否可用并获取设备*
36. if torch.cuda.is\_available():
37. *# 获取 GPU 数量*
38. n\_gpu = torch.cuda.device\_count()
39. print(f"Number of GPUs available: {n\_gpu}")
40. *# 选择用于后续操作的设备*
41. device = torch.device("cuda:0")  *# 可以使用设备 0 作为主设备*
42. model.to(device)  *# 将模型移动到主设备*
43. *# 遍历所有 GPU 并打印信息*
44. for i in range(n\_gpu):
45. gpu\_name = torch.cuda.get\_device\_name(i)
46. print(f"GPU {i}: {gpu\_name}")
47. else:
48. print("CUDA is not available. No GPU detected.")
49. device = torch.device("cpu")  *# 如果没有 GPU，则使用 CPU*
50. *# 增加以下代码以使用 DataParallel*
51. if n\_gpu > 1:
52. logger.info(f"Using {n\_gpu} GPUs")
53. *# 将模型包装在 DataParallel 中*
54. model = torch.nn.DataParallel(model, device\_ids=range(n\_gpu))  *# 使用所有可用的 GPU*
55. else:
56. logger.info(f"Using device: {device}")
57. *# 获取总的训练步数*
58. t\_toal = len(train\_loader) \* args.epochs
59. *# 构建优化器和学习率调度器*
60. optimizer, schedule = build\_optimizer\_and\_scheduler(args, model, t\_toal)
61. *# 初始化训练器*
62. train = Trainer(
63. output\_dir=args.output\_dir,
64. model=model,
65. train\_loader=train\_loader,
66. dev\_loader=dev\_loader,
67. test\_loader=dev\_loader,
68. optimizer=optimizer,
69. schedule=schedule,
70. epochs=args.epochs,
71. device=device,
72. id2label=args.id2label,
73. *# fold=fold\_idx*
74. )
75. *# 训练模型*
76. train.train()

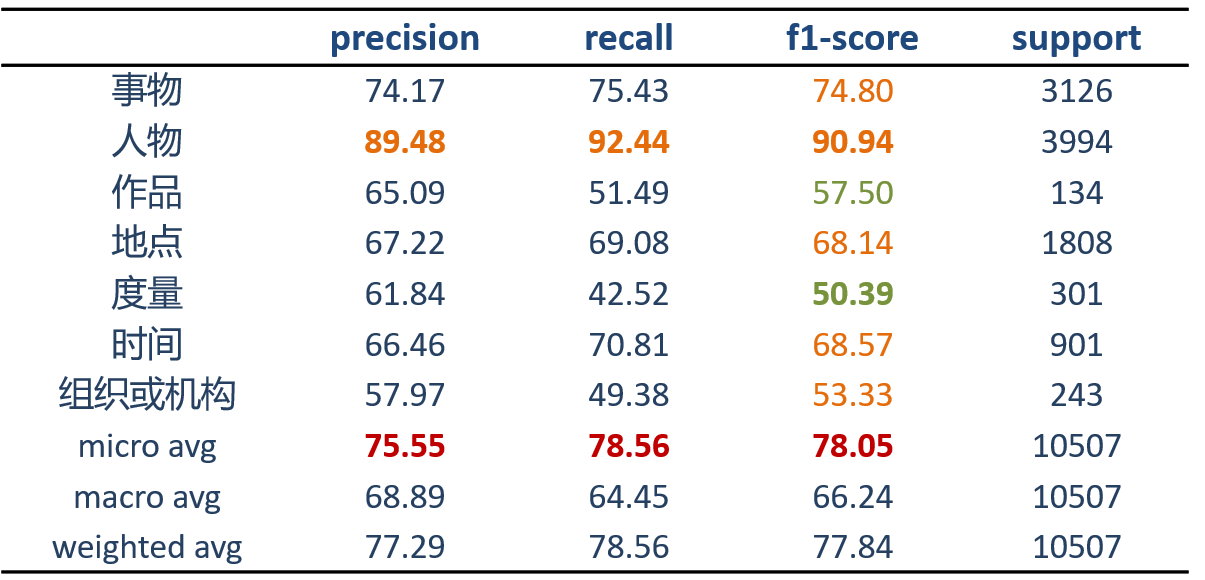
### **4.5模型测试**

验证模型在测试集的效果，生成各种评估数据方便评估模型效率：

1. def test(self):
2. """
3. 测试函数，评估模型在测试集上的性能
4. :return: 返回分类报告，包括精确度、召回率和F1分数等指标
5. """
6. self.best\_model\_path = "best\_model.bin"
7. *# 加载模型*
8. self.model.load\_state\_dict(torch.load(os.path.join(self.output\_dir, self.best\_model\_path)))
9. *# 设置模型为评估模式*
10. self.model.eval()
11. *# 初始化预测和真实标签列表*
12. preds = []
13. trues = []
14. for step, batch\_data in enumerate(tqdm(self.test\_loader)):
15. *# 将批次数据加载到设备上（如 GPU）*
16. for key, value in batch\_data.items():
17. batch\_data[key] = value.to(self.device)
18. *# 获取输入数据、注意力掩码和标签*
19. input\_ids = batch\_data["input\_ids"]
20. attention\_mask = batch\_data["attention\_mask"]
21. labels = batch\_data["labels"]
22. *# 模型前向传播*
23. output = self.model(input\_ids, attention\_mask, labels)
24. *# 获取预测结果*
25. logits = output.logits
26. *# 将预测结果和标签转移到 CPU 上*
27. attention\_mask = attention\_mask.detach().cpu().numpy()
28. labels = labels.detach().cpu().numpy()
29. *# 获取批次大小*
30. batch\_size = input\_ids.size(0)
31. *# 遍历批次里的每个样本*
32. for i in range(batch\_size):
33. *# 获取真实标签的长度*
34. length = sum(attention\_mask[i])
35. *# 获取预测结果，去除 [CLS] 和 [SEP] 标记*
36. logit = logits[i][1:length]
37. *# 将预测结果转换为标签*
38. logit = [self.id2label[i] for i in logit]
39. *# 获取真实标签*
40. label = labels[i][1:length]
41. *# 将真实标签 ID 转换为标签*
42. label = [self.id2label[i] for i in label]
43. *# 存储预测结果和真实标签*
44. preds.append(logit)
45. trues.append(label)
46. *# 生成分类报告，包括精确度、召回率和 F1 分数等指标*
47. report = classification\_report(trues, preds, digits=4)
48. return report

## **（5）实验结果**

1.通过测试生成的各种评估数据如下图所示：



分析：不难看出，在类别样本足够的时候模型的准确率很高，召回率甚至达到了92.44%。平均准确率也接近了70%。但是如果样本不足，例如作品的类别样本只有134，其召回率和得分均低于60%。未来计划通过数据增强技术，补充不足类别的样本以达到更好的测试效果。

2.测试最终生成的实验结果如下图所示：



分析：最终的实验输出满足实验要求的格式。

## **（6）实验总结**

### **6.1实验中遇到的问题以及解决方法**

1.数据分词和标签对齐出错，特别是BERT分词后对标签（如BIO标注）的继承。

问题解决：使用BERT的官方分词器处理数据，确保子词与原始词汇对齐。

2. 训练过程中模型出现过拟合。

问题解决：添加正则化和早停机制；同时使用学习率自动调优。

3. 训练数据中命名实体标签类别分布不平衡，导致测试效果在小规模类别上表现非常不好。

问题解决：采用数据增强，进行实体替换增加小规模类别的样本量。

4.刚开始标记过长，导致绝大部分的句子因为填充不仅影响了实体识别的准确性，占显存过大，同时降低了模型的训练速率

问题解决：通过分析数据集的句子成分，采用更加合理的标签长度。

5.BiLSTM隐藏层维度设置问题，刚开始维度设置太大，导致非常容易出现过拟合问题

问题解决：调试BiLSTM隐藏层维度，最终设置为256层

6.实验刚开始时一直报CRF与BERT冲突

问题解决：版本不兼容问题，更新CRF解决

7.为加快训练速度尝试使用GPU并行技术，但最终因技术难度过高没有落实

8.刚开始模型对人物类型实体的识别准确率特别低

问题解决：最终发现时预训练模型的缺陷，通过小数据集重新训练才能分类人物。

9.小组有尝试使用最新版的RocBERT模型，但因为因为vocab.txt总是报错没用上。

### **6.2实验感想**

在完成基于BERT+BiLSTM+CRF的命名实体识别实验后，我们深刻感受到自然语言处理（NLP）领域的复杂性与挑战，也从中收获了许多研究经验和心得。这不仅是一项技术性的训练，更是一次对科研方法和实践能力的全面考验。下面是我们小组每个人的实验总结：

**1. 王昭**

我主要负责数据预处理和BERT模型微调。通过参与数据预处理和BERT微调的部分，我深刻认识到数据质量对于模型训练的重要性。最初，我遇到了数据标注不一致和标签对齐的问题，特别是在BERT分词后如何确保BIO标签的正确对齐，这给我带来了不少挑战。通过与团队成员的讨论和多次调试，我逐步解决了这些问题，也更加深入地理解了BERT的工作原理。

**2. 王子豪**

我主要负责BiLSTM模型搭建与训练。BiLSTM对于捕捉上下文信息起到了关键作用，但我在设置隐藏层维度和正则化时遇到了困扰。过小的隐藏层维度导致模型无法有效捕捉上下文，过大的维度则造成了过拟合。经过反复调整，我最终找到了一个合适的平衡点。这让我深刻感受到，调参不仅需要理论支持，还需要大量的实验验证。

**3. 贾登辉**

作为负责CRF层的同学，我在实验中深刻体验到CRF在序列标注任务中的关键作用。在开始时，CRF层的训练不稳定，尤其是在标签转移矩阵的初始化上，模型的性能并不理想。通过与小组成员共同分析和调试，我逐渐理解了CRF层的原理，并成功优化了标签转移矩阵的初始化，最终使模型达到了较好的性能。这一过程让我意识到细节决定成败，尤其是模型优化阶段的耐心与细致。

**4. 杨昌霖**

作为负责实验设计和结果评估的成员，我在这次实验中深刻体会到，设计一个合理的实验流程和评估方法是多么重要。在实验过程中，我首先从理论上制定了实验目标和评价指标，然后根据实际情况设计了合理的实验步骤。结果评估阶段，我遇到了模型性能波动较大的问题，通过与团队成员一起分析，发现是标签对齐和数据不平衡造成了影响。在结果分析时，我总结了模型的优缺点，并提出了改进方向，这让我更加理解了模型评估的重要性。

**5.总结**

通过这次实验，深化了我们对NLP领域各种模型的认识，提高了我们的编码能力和项目测评能力。同时也让我们认识到科研不只是解决技术问题，还需要规划实验流程、明确目标并做好时间管理。从文献调研到实验设计，再到问题解决和结果分析，每一步都需要条理清晰、有据可依。这种系统化的思维能力在未来的研究中将起到重要作用。

## **（7）实验分工**

**组长：王昭（负责数据预处理和BERT模型微调）**

数据预处理：

负责原始数据的清洗、分词与BIO标签的对齐。使用BERT分词器进行数据的子词化处理，并确保分词后的子词和标签能够正确对齐。处理数据集中的缺失值、重复数据，并确保标签格式符合BIO标准。

BERT模型微调：

负责加载预训练的BERT模型，并进行微调，以适应命名实体识别任务。并优化模型在特定任务上的表现。评估BERT模型在训练集上的表现，确保模型能够有效地提取上下文信息。

**组员：王子豪（负责BiLSTM模型搭建与训练）**

BiLSTM模型搭建：

负责设计并实现BiLSTM模型，用于捕捉输入序列的上下文信息。配置BiLSTM的层数、隐藏层大小等超参数，以避免过拟合。

BiLSTM模型训练：

负责BiLSTM模型的训练与调优。调整训练过程中使用的超参数（如学习率）来优化模型性能。评估BiLSTM模型的输出，确保其能够有效地捕捉到序列中的上下文信息。

**组员：贾登辉（负责CRF层实现与优化）**

CRF层实现：

负责设计并实现CRF层，用于学习序列标签之间的转移关系。实现CRF层的前向传播和反向传播过程，确保标签的合理转移。

CRF层优化：

负责调试CRF层中的标签转移矩阵初始化，确保模型训练时不会出现数值不稳定问题。通过调整CRF的相关超参数（如正则化强度），提升模型对标签序列的学习能力。监控CRF层的训练过程，确保其与BiLSTM输出能够高效结合，生成最终的标签序列。

**组员：杨昌霖（负责实验设计与结果评估）**

实验设计：

负责设计实验流程和确定评估指标，确保实验可以按计划进行并得出有意义的结果。负责制定实验日志记录标准，跟踪实验过程中的问题及其解决方案。

结果评估与分析：

负责在训练和测试阶段评估模型的表现，包括准确率、召回率、F1值等指标。

进行模型的性能分析，识别瓶颈问题，提出改进方案。