

基于BERT+PET方式文本分类模型搭建

学习目标

- 掌握基于BERT+PET方式模型搭建代码的实现.
- 掌握模型的训练,验证及相关工具代码的实现.
- 掌握使用模型预测代码的实现.

模型搭建

- 本项目中完成BERT+PET模型搭建、训练及应用的步骤如下（注意：因为本项目中使用的是BERT预训练模型，所以直接加载即可，无需重复搭建模型架构）：
 - 一、实现模型工具类函数
 - 二、实现模型训练函数,验证函数
 - 三、实现模型预测函数

一、实现模型工具类函数

- 目的：模型在训练、验证、预测时需要的函数
- 代码路径：/Users/**/PycharmProjects/llm/prompt_tasks/PET/utils
- utils文件夹共包含3个py脚本：verbalizer.py、metirc_utils.py以及common_utils.py

1.1 verbalizer.py

- 目的：定义一个Verbalizer类，用于将一个Label对应到其子Label的映射。
- 导入必备的工具包

```
1 # -*- coding:utf-8 -*-
2 import os
3 from typing import Union, List
4 from pet_config import *
5 pc = ProjectConfig()
```

- 具体实现代码

```
1 class Verbalizer(object):
2     """
3         Verbalizer类，用于将一个Label对应到其子Label的映射。
4     """
```

```
5
6     def __init__(self, verbalizer_file: str, tokenizer, max_label_len:
7         int):
8         """
9             Args:
10                 verbalizer_file (str): verbalizer文件存放地址。
11                 tokenizer: 分词器，用于文本和id之间的转换。
12                 max_label_len (int): 标签长度，若大于则截断，若小于则补齐
13             """
14         self.tokenizer = tokenizer
15         self.label_dict = self.load_label_dict(verbalizer_file)
16         self.max_label_len = max_label_len
17
18     def load_label_dict(self, verbalizer_file: str):
19         """
20             读取本地文件，构建verbalizer字典。
21             Args:
22                 verbalizer_file (str): verbalizer文件存放地址。
23             Returns:
24                 dict -> {
25                     '体育': ['篮球', '足球', '网球', '排球', ...],
26                     '酒店': ['宾馆', '旅馆', '旅店', '酒店', ...],
27                     ...
28                 }
29
30             label_dict = {}
31             with open(verbalizer_file, 'r', encoding='utf8') as f:
32                 for line in f.readlines():
33                     label, sub_labels = line.strip().split('\t')
34                     label_dict[label] = list(set(sub_labels.split(',')))
35             return label_dict
36
37     def find_sub_labels(self, label: Union[list, str]):
38         """
39             通过标签找到对应所有的子标签。
40             Args:
41                 label (Union[list, str]): 标签，文本型 或 id_list, e.g. -> '体育'
42                 or [860, 5509]
43
44             Returns:
45                 dict -> {
46                     'sub_labels': ['足球', '网球'],
47                     'token_ids': [[6639, 4413], [5381, 4413]]
48                 }
49
50             if type(label) == list:    # 如果传入为id_list，则通过tokenizer进行
51                 # 文本转换
52                 while self.tokenizer.pad_token_id in label:
53                     label.remove(self.tokenizer.pad_token_id)
```

```

51         label = ''.join(self.tokenizer.convert_ids_to_tokens(label))
52         # print(f'label-->{label}')
53     if label not in self.label_dict:
54         raise ValueError(f'Label Error: "{label}" not in label_dict')
55
56     sub_labels = self.label_dict[label]
57     ret = {'sub_labels': sub_labels}
58     token_ids = [_id[1:-1] for _id in self.tokenizer(sub_labels)
59     ['input_ids']]
60         # print(f'token_ids-->{token_ids}')
61     for i in range(len(token_ids)):
62         token_ids[i] = token_ids[i][:self.max_label_len] # 对标签进行
截断与补齐
63         if len(token_ids[i]) < self.max_label_len:
64             token_ids[i] = token_ids[i] +
65 [self.tokenizer.pad_token_id] * (self.max_label_len - len(token_ids[i]))
66     ret['token_ids'] = token_ids
67     return ret
68
69 def batch_find_sub_labels(self, label: List[Union[list, str]]):
70     """
71     批量找到子标签。
72
73     Args:
74         label (List[list, str]): 标签列表, [[4510, 5554], [860, 5509]] or
75         ['体育', '电脑']
76
77     Returns:
78         list -> [
79             {
80                 'sub_labels': ['足球', '网球'],
81                 'token_ids': [[6639, 4413], [5381, 4413]]
82             },
83             ...
84         ]
85     """
86
87     return [self.find_sub_labels(l) for l in label]
88
89 def get_common_sub_str(self, str1: str, str2: str):
90     """
91     寻找最大公共子串。
92     str1:abcd
93     str2:ababcbdba
94     """
95
96     lstr1, lstr2 = len(str1), len(str2)
# 生成0矩阵, 为方便后续计算, 比字符串长度多了一列
97     record = [[0 for i in range(lstr2 + 1)] for j in range(lstr1 +
98     1)]
99     p = 0 # 最长匹配对应在str1中的最后一位

```

```

95         maxNum = 0 # 最长匹配长度
96
97         for i in range(lstr1):
98             for j in range(lstr2):
99                 if str1[i] == str2[j]:
100                     record[i+1][j+1] = record[i][j] + 1
101                     if record[i+1][j+1] > maxNum:
102                         maxNum = record[i+1][j+1]
103                         p = i + 1
104
105     return str1[p-maxNum:p], maxNum
106
107
108
109     def hard_mapping(self, sub_label: str):
110         """
111             强匹配函数，当模型生成的子label不存在时，通过最大公共子串找到重合度最高的主
112             label。
113
114             Args:
115                 sub_label (str): 子label。
116
117             Returns:
118                 str: 主label。
119             """
120             label, max_overlap_str = '', 0
121             # print(self.label_dict.items())
122             for main_label, sub_labels in self.label_dict.items():
123                 overlap_num = 0
124                 for s_label in sub_labels: # 求所有子label与当前推理label之间的
125                     overlap_num += self.get_common_sub_str(sub_label,
126 s_label)[1]
127                     if overlap_num >= max_overlap_str:
128                         max_overlap_str = overlap_num
129                         label = main_label
130             return label
131
132     def find_main_label(self, sub_label: Union[list, str],
133                         hard_mapping=True):
134         """
135             通过子标签找到父标签。
136
137             Args:
138                 sub_label (List[Union[list, str]]): 子标签，文本型 或 id_list,
e.g. -> '苹果' or [5741, 3362]
139                 hard_mapping (bool): 当生成的词语不存在时，是否一定要匹配到一个最相似
的label。
140
141             Returns:
142                 str: 父label。
143
144             Raises:
145                 ValueError: 如果没有找到父标签。
146
147             Examples:
148                 self.find_main_label(['苹果'])
149
150                 self.find_main_label([5741, 3362])
151
152                 self.find_main_label(['苹果', '香蕉'])
153
154                 self.find_main_label([5741, 3362, 12345])
155
156                 self.find_main_label([5741, 3362, 12345], hard_mapping=False)
157
158                 self.find_main_label('apple')
159
160                 self.find_main_label([12345])
161
162                 self.find_main_label([12345], hard_mapping=False)
163
164                 self.find_main_label([12345, 56789])
165
166                 self.find_main_label([12345, 56789], hard_mapping=False)
167
168                 self.find_main_label([12345, 56789, 98765])
169
170                 self.find_main_label([12345, 56789, 98765], hard_mapping=False)
171
172                 self.find_main_label([12345, 56789, 98765, 45321])
173
174                 self.find_main_label([12345, 56789, 98765, 45321], hard_mapping=False)
175
176                 self.find_main_label([12345, 56789, 98765, 45321, 32165])
177
178                 self.find_main_label([12345, 56789, 98765, 45321, 32165], hard_mapping=False)
179
180                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435])
181
182                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435], hard_mapping=False)
183
184                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345])
185
186                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345], hard_mapping=False)
187
188                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321])
189
190                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321], hard_mapping=False)
191
192                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216])
193
194                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216], hard_mapping=False)
195
196                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165])
197
198                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165], hard_mapping=False)
199
200                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435])
201
202                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435], hard_mapping=False)
203
204                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345])
205
206                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345], hard_mapping=False)
207
208                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321])
209
210                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321], hard_mapping=False)
211
212                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
213
214                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
215
216                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435])
217
218                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345])
219
220                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345], hard_mapping=False)
221
222                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321])
223
224                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321], hard_mapping=False)
225
226                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
227
228                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216], hard_mapping=False)
229
230                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
231
232                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165], hard_mapping=False)
233
234                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165], hard_mapping=False)
235
236                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435])
237
238                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
239
240                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
241
242                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
243
244                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
245
246                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
247
248                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
249
250                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
251
252                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165], hard_mapping=False)
253
254                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
255
256                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
257
258                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
259
260                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
261
262                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
263
264                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
265
266                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
267
268                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
269
270                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
271
272                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
273
274                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
275
276                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
277
278                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
279
280                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
281
282                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
283
284                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
285
286                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
287
288                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
289
290                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
291
292                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
293
294                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
295
296                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
297
298                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
299
300                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165])
301
302                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
303
304                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
305
306                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
307
308                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
309
310                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
311
312                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
313
314                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
315
316                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
317
318                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
319
320                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
321
322                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
323
324                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
325
326                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
327
328                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
329
330                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216, 32165, 21435, 12345, 54321, 43216])
331
332                 self.find_main_label([12345, 56789, 98765, 45321, 32165, 21435, 123
```

```

138     Returns:
139         dict -> {
140             'label': '水果',
141             'token_ids': [3717, 3362]
142         }
143     """
144     if type(sub_label) == list:      # 如果传入为id_list, 则通过
tokenizer转回来
145     pad_token_id = self.tokenizer.pad_token_id
146     while pad_token_id in sub_label:           # 移除[PAD]token
147         sub_label.remove(pad_token_id)
148     sub_label =
149     ''.join(self.tokenizer.convert_ids_to_tokens(sub_label))
150     # print(sub_label)
151     main_label = '无'
152     for label, s_labels in self.label_dict.items():
153         if sub_label in s_labels:
154             main_label = label
155             break
156
157         if main_label == '无' and hard_mapping:
158             main_label = self.hard_mapping(sub_label)
159             # print(main_label)
160     ret = {
161         'label': main_label,
162         'token_ids': self.tokenizer(main_label)[ 'input_ids' ][1:-1]
163     }
164     return ret
165
166     def batch_find_main_label(self, sub_label: List[Union[list, str]],
hard_mapping=True):
167         """
168         批量通过子标签找父标签。
169
170         Args:
171             sub_label (List[Union[list, str]]): 子标签列表, ['苹果', ...]
172             or [[5741, 3362], ...]
173
174             Returns:
175                 list: [
176                     {
177                         'label': '水果',
178                         'token_ids': [3717, 3362]
179                     },
180                     ...
181                 ]
182             """
183             return [self.find_main_label(l, hard_mapping) for l in sub_label]

```

```
183
184 if __name__ == '__main__':
185     from rich import print
186     from transformers import AutoTokenizer
187
188     tokenizer = AutoTokenizer.from_pretrained(pc.pre_model)
189     verbalizer = Verbalizer(
190         verbalizer_file=pc.verbalizer,
191         tokenizer=tokenizer,
192         max_label_len=2
193     )
194     print(verbalizer.label_dict)
195     # label = [4510, 5554]
196     # ret = verbalizer.find_sub_labels(label)
197     # label = ['电脑', '衣服']
198     label = [[4510, 5554], [6132, 3302]]
199     ret = verbalizer.batch_find_sub_labels(label)
200     print(ret)
```

1.2 common_utils.py

- 目的：定义损失函数、将mask_position位置的token logits转换为token的id。
- 脚本里面包含两个函数：mlm_loss()以及convert_logits_to_ids()
- 导入必备的工具包：

```
1 # coding:utf-8
2 # 导入必备工具包
3 import torch
4 from rich import print
```

- 定义损失函数mlm_loss()

```
1 def mlm_loss(logits, mask_positions, sub_mask_labels,
2               cross_entropy_criterion, device):
3     """
4     计算指定位置的mask token的output与label之间的cross entropy loss。
5
6     Args:
7         logits (torch.tensor): 模型原始输出 -> (batch, seq_len, vocab_size)
8         mask_positions (torch.tensor): mask token的位置 -> (batch,
9         mask_label_num)
10        sub_mask_labels (list): mask token的sub label, 由于每个label的
11        sub_label数目不同, 所以 这里是个变长的list,
12
13        e.g. -> [
14            [[2398, 3352]],
15            [[2398, 3352], [3819, 3861]]]
```

```
13         ]
14     cross_entropy_criterion (CrossEntropyLoss): CE Loss计算器
15     device (str): cpu还是gpu
16
17     Returns:
18     torch.tensor: CE Loss
19     """
20
21     batch_size, seq_len, vocab_size = logits.size()
22     loss = None
23     for single_value in zip(logits, sub_mask_labels, mask_positions):
24         single_logits = single_value[0]
25         single_sub_mask_labels = single_value[1]
26         single_mask_positions = single_value[2]
27
28         # single_mask_logits形状: (mask_label_num, vocab_size)
29         single_mask_logits = single_logits[single_mask_positions]
30
31         # single_mask_logits按照子标签的长度进行复制:
32         # single_mask_logits形状-->(sub_label_num, mask_label_num,
33         vocab_size)
34         single_mask_logits =
35         single_mask_logits.repeat(len(single_sub_mask_labels), 1,
36                                     1)
37
38         #single_mask_logits改变形状: (sub_label_num * mask_label_num,
39         vocab_size)
40         #模型预测的结果
41         single_mask_logits = single_mask_logits.reshape(-1, vocab_size)
42
43         # single_sub_mask_labels形状: (sub_label_num, mask_label_num)
44         single_sub_mask_labels =
45         torch.LongTensor(single_sub_mask_labels).to(device)
46
47         # single_sub_mask_labels形状: # (sub_label_num * mask_label_num)
48         single_sub_mask_labels = single_sub_mask_labels.reshape(-1,
49                                     1).squeeze()
50
51         #if not single_sub_mask_labels.size(): # 处理单token维度下维度缺失的
52         #问题
53         #single_sub_mask_labels =
54         single_sub_mask_labels.unsqueeze(dim=0)
55
56         cur_loss = cross_entropy_criterion(single_mask_logits,
57         single_sub_mask_labels)
58         cur_loss = cur_loss / len(single_sub_mask_labels)
59
60         if not loss:
61             loss = cur_loss
62         else:
```

```
54         loss += cur_loss
55
56     loss = loss / batch_size
57     return loss
```

- 定义convert_logits_to_ids()函数

```
1 def convert_logits_to_ids(
2     logits: torch.tensor,
3     mask_positions: torch.tensor):
4     """
5         输入LM的词表概率分布 (LMModel的logits) , 将mask_position位置的
6         token logits转换为token的id。
7
8     Args:
9         logits (torch.tensor): model output -> (batch, seq_len,
10             vocab_size)
11            mask_positions (torch.tensor): mask token的位置 -> (batch,
12             mask_label_num)
13
14     Returns:
15         torch.LongTensor: 对应mask position上最大概率的推理token -> (batch,
16             mask_label_num)
17         """
18
19     label_length = mask_positions.size()[1] # 标签长度
20     # print(f'label_length--》 {label_length}')
21     batch_size, seq_len, vocab_size = logits.size()
22
23     mask_positions_after_reshaped = []
24
25     for batch, mask_pos in
26         enumerate(mask_positions.detach().cpu().numpy().tolist()):
27         for pos in mask_pos:
28             mask_positions_after_reshaped.append(batch * seq_len + pos)
29
30     # logits形状: (batch_size * seq_len, vocab_size)
31     logits = logits.reshape(batch_size * seq_len, -1)
32
33     # mask_logits形状: (batch * label_num, vocab_size)
34     mask_logits = logits[mask_positions_after_reshaped]
35
36     # predict_tokens形状: (batch * label_num)
37     predict_tokens = mask_logits.argmax(dim=-1)
38
39     # 改变后的predict_tokens形状: (batch, label_num)
40     predict_tokens = predict_tokens.reshape(-1, label_length) # (batch,
41             label_num)
```

```
37     return predict_tokens
```

1.3 metric_utils.py

- 目的：定义（多）分类问题下的指标评估（acc, precision, recall, f1）。
- 导入必备的工具包：

```
1 from typing import List
2
3 import numpy as np
4 import pandas as pd
5 from sklearn.metrics import accuracy_score, precision_score, f1_score
6 from sklearn.metrics import recall_score, confusion_matrix
```

- 定义ClassEvaluator类

```
1 class ClassEvaluator(object):
2
3     def __init__(self):
4         self.goldens = []
5         self.predictions = []
6
7     def add_batch(self, pred_batch: List[List], gold_batch: List[List]):
8         """
9             添加一个batch中的prediction和gold列表，用于后续统一计算。
10
11         Args:
12             pred_batch (list): 模型预测标签列表，e.g. -> [0, 0, 1, 2, 0, ...]
13             or [['体', '育'], ['财', '经'], ...]
14             gold_batch (list): 真实标签标签列表，e.g. -> [1, 0, 1, 2, 0, ...]
15             or [['体', '育'], ['财', '经'], ...]
16             """
17
18         assert len(pred_batch) == len(gold_batch)
19
20         # 若遇到多个子标签构成一个标签的情况
21         if type(gold_batch[0]) in [list, tuple]:
22             # 将所有的label拼接为一个整label: ['体', '育'] -> '体育'
23             pred_batch = [''.join([str(e) for e in ele]) for ele in
24 pred_batch]
25             gold_batch = [''.join([str(e) for e in ele]) for ele in
26 gold_batch]
27
28             self.goldens.extend(gold_batch)
29             self.predictions.extend(pred_batch)
30
31     def compute(self, round_num=2) -> dict:
```

```
27 """
28     根据当前类中累积的变量值，计算当前的P, R, F1。
29
30     Args:
31         round_num (int): 计算结果保留小数点后几位，默认小数点后2位。
32
33     Returns:
34         dict -> {
35             'accuracy': 准确率,
36             'precision': 精准率,
37             'recall': 召回率,
38             'f1': f1值,
39             'class_metrics': {
40                 '0': {
41                     'precision': 该类别下的precision,
42                     'recall': 该类别下的recall,
43                     'f1': 该类别下的f1
44                 },
45                 ...
46             }
47         }
48 """
49     classes, class_metrics, res = sorted(list(set(self.goldens) |
50                                         set(self.predictions)), {}, {})
51
52     # 构建全局指标
53     res['accuracy'] = round(accuracy_score(self.goldens,
54                                         self.predictions), round_num)
55
56     res['precision'] = round(precision_score(self.goldens,
57                                         self.predictions, average='weighted'), round_num)
58
59     # average='weighted'代表：考虑类别的不平衡性，需要计算类别的加权平均。如果是二分类问题则选择参数'binary'
60     res['recall'] = round(recall_score(self.goldens, self.predictions,
61                                         average='weighted'), round_num)
62
63     res['f1'] = round(f1_score(self.goldens, self.predictions,
64                               average='weighted'), round_num)
65
66     try:
67         conf_matrix = np.array(confusion_matrix(self.goldens,
68                                         self.predictions)) # (n_class, n_class)
69         assert conf_matrix.shape[0] == len(classes)
70         for i in range(conf_matrix.shape[0]): # 构建每个class的指标
71             precision = 0 if sum(conf_matrix[:, i]) == 0 else
72             conf_matrix[i, i] / sum(conf_matrix[:, i])
73             recall = 0 if sum(conf_matrix[i, :]) == 0 else
74             conf_matrix[i, i] / sum(conf_matrix[i, :])
```

```
67         f1 = 0 if (precision + recall) == 0 else 2 * precision *
68         recall / (precision + recall)
69         class_metrics[classes[i]] = {
70             'precision': round(precision, round_num),
71             'recall': round(recall, round_num),
72             'f1': round(f1, round_num)
73         }
74     res['class_metrics'] = class_metrics
75 except Exception as e:
76     print(f'[Warning] Something wrong when calculate
77 class_metrics: {e}')
78     print(f'--> goldens: {set(self.goldens)}')
79     print(f'--> predictions: {set(self.predictions)}')
80     print(f'--> diff elements: {set(self.predictions) -
81 set(self.goldens)}')
82     res['class_metrics'] = {}
83
84     return res
85
86
87     def reset(self):
88         """
89             重置积累的数值。
90         """
91         self.goldens = []
92         self.predictions = []
```

二、实现模型训练函数,验证函数

- 目的：实现模型的训练和验证
- 代码路径：/Users/**/PycharmProjects/llm/prompt_tasks/PET/train.py
- 脚本里面包含两个函数：model2train()和evaluate_model()
- 导入必备的工具包

```

1 import os
2 import time
3 from transformers import AutoModelForMaskedLM, AutoTokenizer,
4 get_scheduler
5 from pet_config import *
6 import sys
7 sys.path.append('/Users/ligang/PycharmProjects/llm/prompt_tasks/PET/data_h
8 andle')
9 sys.path.append('/Users/ligang/PycharmProjects/llm/prompt_tasks/PET/utils'
10 )
11 from utils.metirc_utils import ClassEvaluator
12 from utils.common_utils import *
13 from data_handle.data_loader import *
14 from utils.verbalizer import Verbalizer
15 from pet_config import *
16 pc = ProjectConfig()

```

- 定义model2train()函数

```

1 def model2train():
2     model = AutoModelForMaskedLM.from_pretrained(pc.pre_model)
3     tokenizer = AutoTokenizer.from_pretrained(pc.pre_model)
4     verbalizer = Verbalizer(verbalizer_file=pc.verbalizer,
5                             tokenizer=tokenizer,
6                             max_label_len=pc.max_label_len)
7
8     #对参数做权重衰减是为了使函数平滑，然而bias和layernorm的权重参数不影响函数的平滑
9     #性。
10    #他们起到的作用仅仅是缩放平移，因此不需要权重衰减
11    no_decay = [ "bias", "LayerNorm.weight"]
12    optimizer_grouped_parameters = [
13        {
14            "params": [p for n, p in model.named_parameters() if not
15 any(nd in n for nd in no_decay)],
16            "weight_decay": pc.weight_decay,
17        },
18        {
19            "params": [p for n, p in model.named_parameters() if any(nd
20 in n for nd in no_decay)],
21            "weight_decay": 0.0,
22        },
23    ]
24    optimizer = torch.optim.AdamW(optimizer_grouped_parameters,
25 lr=pc.learning_rate)
26    model.to(pc.device)
27
28    train_dataloader, dev_dataloader = get_data()

```

```
25 # 根据训练轮数计算最大训练步数，以便于scheduler动态调整lr
26 num_update_steps_per_epoch = len(train_dataloader)
27 #指定总的训练步数，它会被学习率调度器用来确定学习率的变化规律，确保学习率在整个训
28 练过程中得以合理地调节
29 max_train_steps = pc.epochs * num_update_steps_per_epoch
30 warm_steps = int(pc.warmup_ratio * max_train_steps) # 预热阶段的训练步数
31 lr_scheduler = get_scheduler(
32     name='linear',
33     optimizer=optimizer,
34     num_warmup_steps=warm_steps,
35     num_training_steps=max_train_steps,
36 )
37 loss_list = []
38 tic_train = time.time()
39 metric = ClassEvaluator()
40 criterion = torch.nn.CrossEntropyLoss()
41 global_step, best_f1 = 0, 0
42 print('开始训练：')
43 for epoch in range(pc.epochs):
44     for batch in train_dataloader:
45         logits = model(input_ids=batch['input_ids'].to(pc.device),
46
47             token_type_ids=batch['token_type_ids'].to(pc.device),
48             attention_mask=batch['attention_mask'].to(pc.device)).logits
49             # print(f'模型训练得到的结果logits-->{logits.size()}')
50
51             # 真实标签
52             mask_labels = batch['mask_labels'].numpy().tolist()
53             sub_labels = verbalizer.batch_find_sub_labels(mask_labels)
54             sub_labels = [ele['token_ids'] for ele in sub_labels]
55             # print(f'sub_labels--->{sub_labels}')
56
57             loss = mlm_loss(logits,
58                             batch['mask_positions'].to(pc.device),
59                             sub_labels,
60                             criterion,
61                             pc.device,
62                             )
63             optimizer.zero_grad()
64             loss.backward()
65             optimizer.step()
66             lr_scheduler.step()
67             loss_list.append(float(loss.cpu().detach()))
68             #
69             global_step += 1
70             if global_step % pc.logging_steps == 0:
71                 time_diff = time.time() - tic_train
```

```

71         loss_avg = sum(loss_list) / len(loss_list)
72         print("global step %d, epoch: %d, loss: %.5f, speed: %.2f
73             step/s"
74             % (global_step, epoch, loss_avg, pc.logging_steps /
75                 time_diff))
76         tic_train = time.time()
77
78         if global_step % pc.valid_steps == 0:
79             cur_save_dir = os.path.join(pc.save_dir, "model_%d" %
80                 global_step)
81             if not os.path.exists(cur_save_dir):
82                 os.makedirs(cur_save_dir)
83             model.save_pretrained(os.path.join(cur_save_dir))
84             tokenizer.save_pretrained(os.path.join(cur_save_dir))
85
86             acc, precision, recall, f1, class_metrics =
87             evaluate_model(model,
88
89                 metric,
90
91                 dev_dataloader,
92
93                 tokenizer,
94
95                 verbalizer)
96
97             print("Evaluation precision: %.5f, recall: %.5f, F1:
98                 %.5f" % (precision, recall, f1))
99             if f1 > best_f1:
100                 print(
101                     f"best F1 performance has been updated:
102 {best_f1:.5f} --> {f1:.5f}"
103                     )
104                 print(f'Each Class Metrics are: {class_metrics}')
105                 best_f1 = f1
106                 cur_save_dir = os.path.join(pc.save_dir,
107                     "model_best")
108                 if not os.path.exists(cur_save_dir):
109                     os.makedirs(cur_save_dir)
110                     model.save_pretrained(os.path.join(cur_save_dir))
111                     tokenizer.save_pretrained(os.path.join(cur_save_dir))
112                     tic_train = time.time()
113             print('训练结束')

```

- 定义evaluate_model()函数

```

1 | def evaluate_model(model, metric, data_loader, tokenizer, verbalizer):
2 |     """

```

```

3     在测试集上评估当前模型的训练效果。
4
5     Args:
6         model: 当前模型
7         metric: 评估指标类(metric)
8         data_loader: 测试集的dataloader
9         tokenizer:分词器
10        verbalizer: 标签
11    """
12    model.eval()
13    metric.reset()
14
15    with torch.no_grad():
16        for step, batch in enumerate(data_loader):
17            logits = model(input_ids=batch['input_ids'].to(pc.device),
18
18                token_type_ids=batch['token_type_ids'].to(pc.device),
19
19                attention_mask=batch['attention_mask'].to(pc.device)).logits
20
20                mask_labels = batch['mask_labels'].numpy().tolist() # (batch,
label_num)
21
21                for i in range(len(mask_labels)): # 去掉label中的[PAD] token
22                    while tokenizer.pad_token_id in mask_labels[i]:
23                        mask_labels[i].remove(tokenizer.pad_token_id)
24
24                # id转文字
25                mask_labels = [''.join(tokenizer.convert_ids_to_tokens(t)) for
t in mask_labels]
26
26                # (batch, label_num)
27                predictions = convert_logits_to_ids(logits,
28
28                batch['mask_positions']).cpu().numpy().tolist()
29
29
30
30                # 找到子label属于的主label
31                predictions = verbalizer.batch_find_main_label(predictions)
32                predictions = [ele['label'] for ele in predictions]
33                metric.add_batch(pred_batch=predictions,
34
34                gold_batch=mask_labels)
35                eval_metric = metric.compute()
36                model.train()
37
38
39    return eval_metric['accuracy'], eval_metric['precision'], \
40        eval_metric['recall'], eval_metric['f1'], \
41        eval_metric['class_metrics']

```

- 调用:

```
1 cd /Users/**/PycharmProjects/llm/prompt_tasks/PET
2 # 实现模型训练
3 python train.py
```

- 输出结果:

```
1 .....
2 global step 40, epoch: 4, loss: 0.62105, speed: 1.27 step/s
3 Evaluation precision: 0.78000, recall: 0.77000, F1: 0.76000
4 Each Class Metrics are: {'书籍': {'precision': 0.97, 'recall': 0.82, 'f1': 0.89}, '平板': {'precision': 0.57, 'recall': 0.84, 'f1': 0.68}, '手机': {'precision': 0.0, 'recall': 0.0, 'f1': 0}, '水果': {'precision': 0.95, 'recall': 0.81, 'f1': 0.87}, '洗浴': {'precision': 0.7, 'recall': 0.71, 'f1': 0.7}, '电器': {'precision': 0.0, 'recall': 0.0, 'f1': 0}, '电脑': {'precision': 0.86, 'recall': 0.38, 'f1': 0.52}, '蒙牛': {'precision': 1.0, 'recall': 0.68, 'f1': 0.81}, '衣服': {'precision': 0.71, 'recall': 0.91, 'f1': 0.79}, '酒店': {'precision': 1.0, 'recall': 0.88, 'f1': 0.93}}
5 global step 50, epoch: 6, loss: 0.50076, speed: 1.23 step/s
6 global step 60, epoch: 7, loss: 0.41744, speed: 1.23 step/s
7 ...
8 global step 390, epoch: 48, loss: 0.06674, speed: 1.20 step/s
9 global step 400, epoch: 49, loss: 0.06507, speed: 1.21 step/s
10 Evaluation precision: 0.78000, recall: 0.76000, F1: 0.75000
```

- 结论: BERT+PET模型在训练集上的表现是精确率=78%
- 注意: 本项目中只用了60条样本, 在接近600条样本上精确率就已经达到了78%, 如果想让指标更高, 可以扩增样本。

三、实现模型预测函数

- 目的: 加载训练好的模型并测试效果
- 代码路径: /Users/**/PycharmProjects/llm/prompt_tasks/PET/inference.py
- 导入必备的工具包

```

1 import time
2 from typing import List
3
4 import torch
5 from rich import print
6 from transformers import AutoTokenizer, AutoModelForMaskedLM
7 import sys
8 sys.path.append('/Users/**/PycharmProjects/llm/prompt_tasks/PET/data_handle')
9 sys.path.append('/Users/**/PycharmProjects/llm/prompt_tasks/PET/utils')
10 from utils.verbalizer import Verbalizer
11 from data_handle.template import HardTemplate
12 from data_handle.data_preprocess import convert_example
13 from utils.common_utils import convert_logits_to_ids

```

- 预测代码具体实现

```

1 device = 'mps:0'
2 # device='cuda:0'
3 model_path = 'checkpoints/model_best'
4 tokenizer = AutoTokenizer.from_pretrained(model_path)
5 model = AutoModelForMaskedLM.from_pretrained(model_path)
6 model.to(device).eval()
7
8 max_label_len = 2                                # 标签最大长度
9 verbalizer = Verbalizer(
10     verbalizer_file='data/verbalizer.txt',
11     tokenizer=tokenizer,
12     max_label_len=max_label_len
13 )
14 prompt = open('data/prompt.txt',
15                 'r', encoding='utf8').readlines()[0].strip()    # prompt定义
16 hard_template = HardTemplate(prompt=prompt)          # 模板转换器定义
17 print(f'Prompt is -> {prompt}')
18
19
20 def inference(contents: List[str]):
21     """
22         推理函数，输入原始句子，输出mask label的预测值。
23
24     Args:
25         contents (List[str]): 描原始句子列表。
26     """
27     with torch.no_grad():
28         start_time = time.time()
29         examples = {'text': contents}

```

```
30     tokenized_output = convert_example(
31         examples,
32         tokenizer,
33         hard_template=hard_template,
34         max_seq_len=128,
35         max_label_len=max_label_len,
36         train_mode=False,
37         return_tensor=True
38     )
39     logits = model(input_ids=tokenized_output['input_ids'].to(device),
40
41         token_type_ids=tokenized_output['token_type_ids'].to(device),
42
43         attention_mask=tokenized_output['attention_mask'].to(device)).logits
44     predictions = convert_logits_to_ids(logits,
45     tokenized_output['mask_positions']).cpu().numpy().tolist() # (batch,
46     label_num)
47
48     # 找到子label属于的主label
49     predictions = verbalizer.batch_find_main_label(predictions)
50
51
52
53 if __name__ == '__main__':
54     contents = [
55         '天台很好看，躺在躺椅上很悠闲，因为活动所以我觉得性价比还不错，适合一家出行，特别是去迪士尼也蛮近的，下次有机会肯定还会再来的，值得推荐',
56         '环境，设施，很棒，周边配套设施齐全，前台小姐姐超级漂亮！酒店很赞，早餐不错，服务态度很好，前台美眉很漂亮。性价比超高的一家酒店。强烈推荐',
57         "物流超快，隔天就到了，还没用，屯着出游的时候用的，听方便的，占地小",
58         "福行市来到无早集市，因为是喜欢的面包店，所以跑来集市看看。第一眼就看到了，之前在微店买了小刘，这次买了老刘，还有一直喜欢的巧克力磅蛋糕。好奇老板为啥不做柠檬磅蛋糕了，微店一直都是买不到的状态。因为不爱碱水硬欧之类的，所以期待老板多来点其他小点，饼干一直是大爱，那天好像也没看到",
59         "服务很用心，房型也很舒服，小朋友很喜欢，下次去嘉定还会再选择。床铺柔软舒适，晚上休息很安逸，隔音效果不错赞，下次还会来"
60     ]
61     print("针对下面的文本评论，请分别给出对应所属类别：")
62     res = inference(contents)
63     #print('inference label(s):', res)
64     new_dict = {}
65     for i in range(len(contents)):
66         new_dict[contents[i]] = res[i]
67     print(new_dict)
```

- 结果展示

```
1  {
2      '天台很好看，躺在躺椅上很悠闲，因为活动所以我觉得性价比还不错，适合一家出
3      行，特别是去迪士尼也蛮近的，下次有机会肯定还会再来的，值得推荐'：'酒店'，
4      '环境，设施，很棒，周边配套设施齐全，前台小姐姐超级漂亮！酒店很赞，早餐不
5      错，服务态度很好，前台美眉很漂亮。性价比超高的一家酒店。强烈推荐'：'酒店'，
6      '物流超快，隔天就到了，还没用，屯着出游的时候用的，听方便的，占地小'：'平板'，
7      '福行市来到无早集市，因为是喜欢的面包店，所以跑来集市看看。第一眼就看到了
8      ，之前在微店买了小刘，这次买了老刘，还有一直喜欢的巧克力磅蛋糕。好奇老板为啥不做
9      柠檬磅蛋糕了，微店一直都是买不到的状态。因为不爱碱水硬欧之类的，所以期待老板多来
10     点其他小点，饼干一直也是大爱，那天好像也没看到'：'水果'，
11     '服务很用心，房型也很舒服，小朋友很喜欢，下次去嘉定还会再选择。床铺柔软舒
12     适，晚上休息很安逸，隔音效果不错赞，下次还会来'：'酒店'
13 }
```

小节总结

- 本小节实现了基于BERT+PET模型的构建，并完成了训练和测试评估。
-
-
-
-