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

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

- 具体实现代码

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

```python

    def __init__(self, verbalizer_file: str, tokenizer, max_label_len: int):
        """
        Args:
            verbalizer_file (str): verbalizer文件存放地址。
            tokenizer: 分词器，用于文本和id之间的转换。
            max_label_len (int): 标签长度，若大于则截断，若小于则补齐
        """
        self.tokenizer = tokenizer
        self.label_dict = self.load_label_dict(verbalizer_file)
        self.max_label_len = max_label_len

    def load_label_dict(self, verbalizer_file: str):
        """
        读取本地文件，构建verbalizer字典。
        Args:
            verbalizer_file (str): verbalizer文件存放地址。
        Returns:
            dict -> {
                '体育': ['篮球', '足球','网球', '排球',  ...],
                '酒店': ['宾馆', '旅馆', '旅店', '酒店', ...],
                ...
            }
        """
        label_dict = {}
        with open(verbalizer_file, 'r', encoding='utf8') as f:
            for line in f.readlines():
                label, sub_labels = line.strip().split('\t')
                label_dict[label] = list(set(sub_labels.split(',')))
        return label_dict

    def find_sub_labels(self, label: Union[list, str]):
        """
        通过标签找到对应所有的子标签。
      Args:
        label (Union[list, str]): 标签，文本型 或 id_list，e.g. -> '体育' or [860, 5509]

      Returns:
            dict -> {
                'sub_labels': ['足球', '网球'],
                'token_ids': [[6639, 4413], [5381, 4413]]
            }
        """
        if type(label) == list:    # 如果传入为id_list，则通过tokenizer进行文本转换
            while self.tokenizer.pad_token_id in label:
                label.remove(self.tokenizer.pad_token_id)
```

```
            label = ''.join(self.tokenizer.convert_ids_to_tokens(label))
        # print(f'label-->{label}')
        if label not in self.label_dict:
            raise ValueError(f'Lable Error: "{label}" not in label_dict')

        sub_labels = self.label_dict[label]
        ret = {'sub_labels': sub_labels}
        token_ids = [_id[1:-1] for _id in self.tokenizer(sub_labels)['input_ids']]
        # print(f'token_ids-->{token_ids}')
        for i in range(len(token_ids)):
            token_ids[i] = token_ids[i][:self.max_label_len]  # 对标签进行截断与补齐
            if len(token_ids[i]) < self.max_label_len:
                token_ids[i] = token_ids[i] + [self.tokenizer.pad_token_id] * (self.max_label_len - len(token_ids[i]))
        ret['token_ids'] = token_ids
        return ret

    def batch_find_sub_labels(self, label: List[Union[list, str]]):
        """
        批量找到子标签。

        Args:
        label (List[list, str]): 标签列表, [[4510, 5554], [860, 5509]] or ['体育', '电脑']

        Returns:
            list -> [
                        {
                         'sub_labels': ['足球', '网球'],
                         'token_ids': [[6639, 4413], [5381, 4413]]
                        },
                        ...
                    ]
        """
        return [self.find_sub_labels(l) for l in label]

    def get_common_sub_str(self, str1: str, str2: str):
        """
        寻找最大公共子串。
        str1:abcd
        str2:abadbcdba
        """
        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
        record = [[0 for i in range(lstr2 + 1)] for j in range(lstr1 + 1)]
        p = 0  # 最长匹配对应在str1中的最后一位
```

```
        maxNum = 0  # 最长匹配长度

        for i in range(lstr1):
            for j in range(lstr2):
                if str1[i] == str2[j]:
                    record[i+1][j+1] = record[i][j] + 1
                    if record[i+1][j+1] > maxNum:
                        maxNum = record[i+1][j+1]
                        p = i + 1

        return str1[p-maxNum:p], maxNum



    def hard_mapping(self, sub_label: str):
        """
        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label。

        Args:
            sub_label (str): 子label。

        Returns:
            str: 主label。
        """
        label, max_overlap_str = '', 0
        # print(self.label_dict.items())
        for main_label, sub_labels in self.label_dict.items():
            overlap_num = 0
            for s_label in sub_labels:  # 求所有子label与当前推理label之间的最长公共子串长度
                overlap_num += self.get_common_sub_str(sub_label, s_label)[1]
            if overlap_num >= max_overlap_str:
                max_overlap_str = overlap_num
                label = main_label
        return label

    def find_main_label(self, sub_label: Union[list, str], hard_mapping=True):
        """
        通过子标签找到父标签。

        Args:
            sub_label (List[Union[list, str]]): 子标签，文本型 或 id_list, e.g. -> '苹果' or [5741, 3362]
            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。

```

```
        Returns:
            dict -> {
                'label': '水果',
                'token_ids': [3717, 3362]
            }
        """
        if type(sub_label) == list:     # 如果传入为id_list, 则通过tokenizer转回来
            pad_token_id = self.tokenizer.pad_token_id
            while pad_token_id in sub_label:           # 移除[PAD]token
                sub_label.remove(pad_token_id)
            sub_label = ''.join(self.tokenizer.convert_ids_to_tokens(sub_label))
        # print(sub_label)
        main_label = '无'
        for label, s_labels in self.label_dict.items():
            if sub_label in s_labels:
                main_label = label
                break

        if main_label == '无' and hard_mapping:
            main_label = self.hard_mapping(sub_label)
        # print(main_label)
        ret = {
            'label': main_label,
            'token_ids': self.tokenizer(main_label)['input_ids'][1:-1]
        }
        return ret

    def batch_find_main_label(self, sub_label: List[Union[list, str]], hard_mapping=True):
        """
        批量通过子标签找父标签。

        Args:
            sub_label (List[Union[list, str]]): 子标签列表, ['苹果', ...] or [[5741, 3362], ...]

        Returns:
            list: [
                    {
                    'label': '水果',
                    'token_ids': [3717, 3362]
                    },
                    ...
            ]
        """
        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,
   mask_label_num)
9          sub_mask_labels (list): mask token的sub label, 由于每个label的
   sub_label数目不同，所以  这里是个变长的list,
10                                     e.g. -> [
11                                         [[2398, 3352]],
12                                         [[2398, 3352], [3819, 3861]]
```

```
                                    ]
        cross_entropy_criterion (CrossEntropyLoss): CE Loss计算器
        device (str): cpu还是gpu

    Returns:
        torch.tensor: CE Loss
    """
    batch_size, seq_len, vocab_size = logits.size()
    loss = None
    for single_value in zip(logits, sub_mask_labels, mask_positions):
        single_logits = single_value[0]
        single_sub_mask_labels = single_value[1]
        single_mask_positions = single_value[2]

        # single_mask_logits形状：(mask_label_num, vocab_size)
        single_mask_logits = single_logits[single_mask_positions]

        # single_mask_logits按照子标签的长度进行复制：
        # single_mask_logits形状-->(sub_label_num, mask_label_num,
vocab_size)
        single_mask_logits =
single_mask_logits.repeat(len(single_sub_mask_labels), 1,
                                                          1)

        #single_mask_logits改变形状：(sub_label_num * mask_label_num,
vocab_size)
        #模型预测的结果
        single_mask_logits = single_mask_logits.reshape(-1, vocab_size)

        # single_sub_mask_labels形状：(sub_label_num, mask_label_num)
        single_sub_mask_labels =
torch.LongTensor(single_sub_mask_labels).to(device)

        # single_sub_mask_labels形状：  # (sub_label_num * mask_label_num)
        single_sub_mask_labels = single_sub_mask_labels.reshape(-1,
1).squeeze()

        #if not single_sub_mask_labels.size():  # 处理单token维度下维度缺失的
问题
            #single_sub_mask_labels =
single_sub_mask_labels.unsqueeze(dim=0)

        cur_loss = cross_entropy_criterion(single_mask_logits,
single_sub_mask_labels)
        cur_loss = cur_loss / len(single_sub_mask_labels)

        if not loss:
            loss = cur_loss
        else:
```

```
            loss += cur_loss

    loss = loss / batch_size
    return loss
```

- 定义convert_logits_to_ids()函数

```
def convert_logits_to_ids(
        logits: torch.tensor,
        mask_positions: torch.tensor):
    """
    输入LM的词表概率分布（LMModel的logits），将mask_position位置的
    token logits转换为token的id。

    Args:
        logits (torch.tensor): model output -> (batch, seq_len, vocab_size)
        mask_positions (torch.tensor): mask token的位置 -> (batch, mask_label_num)

    Returns:
        torch.LongTensor: 对应mask position上最大概率的推理token -> (batch, mask_label_num)
    """
    label_length = mask_positions.size()[1]  # 标签长度
    # print(f'label_length--》{label_length}')
    batch_size, seq_len, vocab_size = logits.size()

    mask_positions_after_reshaped = []

    for batch, mask_pos in enumerate(mask_positions.detach().cpu().numpy().tolist()):
        for pos in mask_pos:
            mask_positions_after_reshaped.append(batch * seq_len + pos)

    # logits形状：(batch_size * seq_len, vocab_size)
    logits = logits.reshape(batch_size * seq_len, -1)

    # mask_logits形状：(batch * label_num, vocab_size)
    mask_logits = logits[mask_positions_after_reshaped]

    # predict_tokens形状：  (batch * label_num)
    predict_tokens = mask_logits.argmax(dim=-1)

    # 改变后的predict_tokens形状：  (batch, label_num)
    predict_tokens = predict_tokens.reshape(-1, label_length)  # (batch, label_num)

```

```
    return predict_tokens
```

1.3 metirc_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：

```
from typing import List

import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score, precision_score, f1_score
from sklearn.metrics import recall_score, confusion_matrix
```

- 定义ClassEvaluator类

```
class ClassEvaluator(object):

    def __init__(self):
        self.goldens = []
        self.predictions = []

    def add_batch(self, pred_batch: List[List], gold_batch: List[List]):
        """
        添加一个batch中的prediction和gold列表，用于后续统一计算。

        Args:
            pred_batch (list): 模型预测标签列表, e.g. -> [0, 0, 1, 2, 0, ...] or [['体', '育'], ['财', '经'], ...]
            gold_batch (list): 真实标签标签列表, e.g. -> [1, 0, 1, 2, 0, ...] or [['体', '育'], ['财', '经'], ...]
        """
        assert len(pred_batch) == len(gold_batch)

        # 若遇到多个子标签构成一个标签的情况
        if type(gold_batch[0]) in [list, tuple]:
            # 将所有的label拼接为一个整label: ['体', '育'] -> '体育'
            pred_batch = [''.join([str(e) for e in ele]) for ele in pred_batch]
            gold_batch = [''.join([str(e) for e in ele]) for ele in gold_batch]

        self.goldens.extend(gold_batch)
        self.predictions.extend(pred_batch)

    def compute(self, round_num=2) -> dict:
```

```
        """
        根据当前类中累积的变量值，计算当前的P，R，F1。

        Args:
            round_num (int): 计算结果保留小数点后几位，默认小数点后2位。

        Returns:
            dict -> {
                'accuracy': 准确率,
                'precision': 精准率,
                'recall': 召回率,
                'f1': f1值,
                'class_metrics': {
                    '0': {
                            'precision': 该类别下的precision,
                            'recall': 该类别下的recall,
                            'f1': 该类别下的f1
                        },
                    ...
                }
            }
        """
        classes, class_metrics, res = sorted(list(set(self.goldens) | set(self.predictions))), {}, {}

        # 构建全局指标
        res['accuracy'] = round(accuracy_score(self.goldens, self.predictions), round_num)

        res['precision'] = round(precision_score(self.goldens, self.predictions, average='weighted'), round_num)

        # 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’
        res['recall'] = round(recall_score(self.goldens, self.predictions, average='weighted'), round_num)

        res['f1'] = round(f1_score(self.goldens, self.predictions, average='weighted'), round_num)

        try:
            conf_matrix = np.array(confusion_matrix(self.goldens, self.predictions))  # (n_class, n_class)
            assert conf_matrix.shape[0] == len(classes)
            for i in range(conf_matrix.shape[0]):  # 构建每个class的指标
                precision = 0 if sum(conf_matrix[:, i]) == 0 else conf_matrix[i, i] / sum(conf_matrix[:, i])
                recall = 0 if sum(conf_matrix[i, :]) == 0 else conf_matrix[i, i] / sum(conf_matrix[i, :])
```

```
                f1 = 0 if (precision + recall) == 0 else 2 * precision * recall / (precision + recall)
                class_metrics[classes[i]] = {
                    'precision': round(precision, round_num),
                    'recall': round(recall, round_num),
                    'f1': round(f1, round_num)
                }
            res['class_metrics'] = class_metrics
        except Exception as e:
            print(f'[Warning] Something wrong when calculate class_metrics: {e}')
            print(f'-> goldens: {set(self.goldens)}')
            print(f'-> predictions: {set(self.predictions)}')
            print(f'-> diff elements: {set(self.predictions) - set(self.goldens)}')
            res['class_metrics'] = {}

        return res

    def reset(self):
        """
        重置积累的数值。
        """
        self.goldens = []
        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

```
import os
import time
from transformers import AutoModelForMaskedLM, AutoTokenizer,
get_scheduler
from pet_config import *
import sys
sys.path.append('/Users/ligang/PycharmProjects/llm/prompt_tasks/PET/data_h
andle')
sys.path.append('/Users/ligang/PycharmProjects/llm/prompt_tasks/PET/utils'
)
from utils.metirc_utils import ClassEvaluator
from utils.common_utils import *
from data_handle.data_loader import *
from utils.verbalizer import Verbalizer
from pet_config import *
pc = ProjectConfig()
```

- 定义model2train()函数

```
def model2train():
    model = AutoModelForMaskedLM.from_pretrained(pc.pre_model)
    tokenizer = AutoTokenizer.from_pretrained(pc.pre_model)
    verbalizer = Verbalizer(verbalizer_file=pc.verbalizer,
                            tokenizer=tokenizer,
                            max_label_len=pc.max_label_len)

    #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
性。
    #他们起到的作用仅仅是缩放平移，因此不需要权重衰减
    no_decay = ["bias", "LayerNorm.weight"]
    optimizer_grouped_parameters = [
        {
            "params": [p for n, p in model.named_parameters() if not
any(nd in n for nd in no_decay)],
            "weight_decay": pc.weight_decay,
        },
        {
            "params": [p for n, p in model.named_parameters() if any(nd
in n for nd in no_decay)],
            "weight_decay": 0.0,
        },
    ]
    optimizer = torch.optim.AdamW(optimizer_grouped_parameters,
lr=pc.learning_rate)
    model.to(pc.device)

    train_dataloader, dev_dataloader = get_data()
```

```
    # 根据训练轮数计算最大训练步数，以便于scheduler动态调整lr
    num_update_steps_per_epoch = len(train_dataloader)
    #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练过程中得以合理地调节
    max_train_steps = pc.epochs * num_update_steps_per_epoch
    warm_steps = int(pc.warmup_ratio * max_train_steps) # 预热阶段的训练步数
    lr_scheduler = get_scheduler(
        name='linear',
        optimizer=optimizer,
        num_warmup_steps=warm_steps,
        num_training_steps=max_train_steps,
    )

    loss_list = []
    tic_train = time.time()
    metric = ClassEvaluator()
    criterion = torch.nn.CrossEntropyLoss()
    global_step, best_f1 = 0, 0
    print('开始训练：')
    for epoch in range(pc.epochs):
        for batch in train_dataloader:
            logits = model(input_ids=batch['input_ids'].to(pc.device),

token_type_ids=batch['token_type_ids'].to(pc.device),

attention_mask=batch['attention_mask'].to(pc.device)).logits
            # print(f'模型训练得到的结果logits-->{logits.size()}')

            # 真实标签
            mask_labels = batch['mask_labels'].numpy().tolist()
            sub_labels = verbalizer.batch_find_sub_labels(mask_labels)
            sub_labels = [ele['token_ids'] for ele in sub_labels]
            # print(f'sub_labels--->{sub_labels}')

            loss = mlm_loss(logits,
                            batch['mask_positions'].to(pc.device),
                            sub_labels,
                            criterion,
                            pc.device,
                            )
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            lr_scheduler.step()
            loss_list.append(float(loss.cpu().detach()))
            # #
            global_step += 1
            if global_step % pc.logging_steps == 0:
                time_diff = time.time() - tic_train
```

```
                loss_avg = sum(loss_list) / len(loss_list)
                print("global step %d, epoch: %d, loss: %.5f, speed: %.2f step/s"
                      % (global_step, epoch, loss_avg, pc.logging_steps / time_diff))
                tic_train = time.time()

            if global_step % pc.valid_steps == 0:
                cur_save_dir = os.path.join(pc.save_dir, "model_%d" % global_step)
                if not os.path.exists(cur_save_dir):
                    os.makedirs(cur_save_dir)
                model.save_pretrained(os.path.join(cur_save_dir))
                tokenizer.save_pretrained(os.path.join(cur_save_dir))

                acc, precision, recall, f1, class_metrics = evaluate_model(model,

  metric,

 dev_dataloader,

  tokenizer,

  verbalizer)

                print("Evaluation precision: %.5f, recall: %.5f, F1: %.5f" % (precision, recall, f1))
                if f1 > best_f1:
                    print(
                        f"best F1 performence has been updated: {best_f1:.5f} --> {f1:.5f}"
                    )
                    print(f'Each Class Metrics are: {class_metrics}')
                    best_f1 = f1
                    cur_save_dir = os.path.join(pc.save_dir, "model_best")
                    if not os.path.exists(cur_save_dir):
                        os.makedirs(cur_save_dir)
                    model.save_pretrained(os.path.join(cur_save_dir))
                    tokenizer.save_pretrained(os.path.join(cur_save_dir))
                tic_train = time.time()
    print('训练结束')
```

- 定义evaluate_model()函数

```
def evaluate_model(model, metric, data_loader, tokenizer, verbalizer):
    """
```

```
    在测试集上评估当前模型的训练效果。

    Args:
        model: 当前模型
        metric: 评估指标类(metric)
        data_loader: 测试集的dataloader
        tokenizer:分词器
        verbalizer: 标签
    """
    model.eval()
    metric.reset()

    with torch.no_grad():
        for step, batch in enumerate(data_loader):
            logits = model(input_ids=batch['input_ids'].to(pc.device),
token_type_ids=batch['token_type_ids'].to(pc.device),
attention_mask=batch['attention_mask'].to(pc.device)).logits
            mask_labels = batch['mask_labels'].numpy().tolist()  # (batch,
label_num)
            for i in range(len(mask_labels)):  # 去掉label中的[PAD] token
                while tokenizer.pad_token_id in mask_labels[i]:
                    mask_labels[i].remove(tokenizer.pad_token_id)

            # id转文字
            mask_labels = [''.join(tokenizer.convert_ids_to_tokens(t)) for
t in mask_labels]

              # (batch, label_num)
            predictions = convert_logits_to_ids(logits,
batch['mask_positions']).cpu().numpy().tolist()

            # 找到子label属于的主label
            predictions = verbalizer.batch_find_main_label(predictions)
            predictions = [ele['label'] for ele in predictions]
            metric.add_batch(pred_batch=predictions,
gold_batch=mask_labels)
    eval_metric = metric.compute()
    model.train()

    return eval_metric['accuracy'], eval_metric['precision'], \
           eval_metric['recall'], eval_metric['f1'], \
           eval_metric['class_metrics']
```

- 调用:

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

- 输出结果:

```
.....
global step 40, epoch: 4, loss: 0.62105, speed: 1.27 step/s
Evaluation precision: 0.78000, recall: 0.77000, F1: 0.76000
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}}
global step 50, epoch: 6, loss: 0.50076, speed: 1.23 step/s
global step 60, epoch: 7, loss: 0.41744, speed: 1.23 step/s
...
global step 390, epoch: 48, loss: 0.06674, speed: 1.20 step/s
global step 400, epoch: 49, loss: 0.06507, speed: 1.21 step/s
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

```python
import time
from typing import List

import torch
from rich import print
from transformers import AutoTokenizer, AutoModelForMaskedLM
import sys
sys.path.append('/Users/**/PycharmProjects/llm/prompt_tasks/PET/data_handle')
sys.path.append('/Users/**/PycharmProjects/llm/prompt_tasks/PET/utils')
from utils.verbalizer import Verbalizer
from data_handle.template import HardTemplate
from data_handle.data_preprocess import convert_example
from utils.common_utils import convert_logits_to_ids
```

- 预测代码具体实现

```python
device = 'mps:0'
# device='cuda:0'
model_path = 'checkpoints/model_best'
tokenizer = AutoTokenizer.from_pretrained(model_path)
model = AutoModelForMaskedLM.from_pretrained(model_path)
model.to(device).eval()

max_label_len = 2                                   # 标签最大长度
verbalizer = Verbalizer(
        verbalizer_file='data/verbalizer.txt',
        tokenizer=tokenizer,
        max_label_len=max_label_len
    )
prompt = open('data/prompt.txt',
                'r', encoding='utf8').readlines()[0].strip()    # prompt定义
hard_template = HardTemplate(prompt=prompt)                                 # 模板转换器定义
print(f'Prompt is -> {prompt}')


def inference(contents: List[str]):
    """
    推理函数，输入原始句子，输出mask label的预测值。

    Args:
        contents (List[str]): 描原始句子列表。
    """
    with torch.no_grad():
        start_time = time.time()
        examples = {'text': contents}
```

```
        tokenized_output = convert_example(
            examples,
            tokenizer,
            hard_template=hard_template,
            max_seq_len=128,
            max_label_len=max_label_len,
            train_mode=False,
            return_tensor=True
        )
        logits = model(input_ids=tokenized_output['input_ids'].to(device),

 token_type_ids=tokenized_output['token_type_ids'].to(device),

 attention_mask=tokenized_output['attention_mask'].to(device)).logits
        predictions = convert_logits_to_ids(logits,
tokenized_output['mask_positions']).cpu().numpy().tolist()  # (batch,
label_num)

        # 找到子label属于的主label
        predictions = verbalizer.batch_find_main_label(predictions)

        predictions = [ele['label'] for ele in predictions]
        used = time.time() - start_time
        print(f'Used {used}s.')
        return predictions


if __name__ == '__main__':
    contents = [
        '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',
        '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',
        "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",
        "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，饼干一直也
是大爱，那天好像也没看到",
        "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"
    ]
    print("针对下面的文本评论，请分别给出对应所属类别：")
    res = inference(contents)
    #print('inference label(s):', res)
    new_dict = {}
    for i in range(len(contents)):
        new_dict[contents[i]] = res[i]
    print(new_dict)
```

- 结果展示

```
{
    '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': '酒店',
    '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': '酒店',
    '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',
    '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，饼干一直也是大爱，那天好像也没看到': '水果',
    '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': '酒店'
}
```

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.