自然语言处理

实验二：面向目标的情感分类

201300035 方盛俊 人工智能学院

目录

一、 方案设计

本次任务要求预测目标对象在句子中的情感极性，一个句子可能包含多个目标对象, 每个目标会构成单独的样例。情感极性分为三类, 1 表示 positive, 0 表示 neutral, -1 表示 negative。

相比于上一次的整句情感分类任务，本次任务的难点在于如何在情感分类时，对预测目标的影响进行建模。经过分析，我采用了依存句法树生成权重 + BERT 分类模型的方式来完成本次任务，采用了如下方案：

1. 数据预处理：
   1. 对输入数据进行清洗，如将 `'-LRB-'` 替换回 `'('`，方便进一步进行句法分析；
   2. 使用 `stanza` 库的 Dependency Parsing 模块生成依存句法树；
   3. target 可能对应多个 tokens，需要寻找离 root 最近的 token；
   4. 基于依存句法树构，找到 target root token 与其他 tokens 之间的距离 `dists`；
   5. 对 `dists` 应用函数 $h(x) = 1 / x^2$ 生成每个 tokens 对应权重 `weights`；
2. BERT 分类模型分类：
   1. 使用 `transformers` 库的 `BertTokenizer` 对输入数据进行划分；
   2. 加入额外信息 `QUESTION` 形成 `[CLS] SENTENCE [SEP] QUESTION [SEP]` 输入, 其中 `QUESTION` 为 `"What is the sentiment class of {target} in the sentence?"`，相当于对分类任务的一种提示；
   3. 使用 `transformers` 库的 `BertModel` 搭建了一个 BERT 分类模型；
   4. 在数据经过 BERT 输出后，得到 `last_hidden_state`，并通过预处理得到的 `weights` 进行加权平均，得到消去 `seq_len` 维的结果 `weighted_output`；
   5. 使用线性层 `nn.Linear(768, num_labels)` 对输出结果进行分类。
3. 使用 PyTorch 的梯度下降对模型进行训练与微调。

代码结构如下：

1. `preprocess.py`：数据预处理文件，包含数据清洗、依存句法树分析、权重生成。
2. `sentiment-analysis-bert.ipynb`：BERT 分类模型的构建、训练以及最终结果生成。
3. `sentiment-analysis-rnn.ipynb`：基于 LSTM 的失败尝试。

二、 模块描述

2.1 全局参数

由于我的本地机器缺乏高效的 GPU, 代码文件使用了 Colab 来训练, 以利用 GPU 加快训练过程。为了统一 Colab 和本地机器的参数, 我将所有的全局参数都放在了 `config` 的字典参数中，这样在 Colab 和本地机器上运行代码时，以及调节超参数时，只需要动态修改 `config` 的字典参数中的参数即可。

```
# 一些全局配置
config = {
```

```
    'is_train': True,  # 是否进行训练
    'is_save': True,  # 是否保存模型文件
    'is_load': True,  # 是否加载模型文件
    'is_save_result': True,  # 是否保存结果

    # ...

    # 训练数据划分相关配置
    'random_seed': 42,  # 随机种子
    'train_set_ratio': 0.99,  # 训练集占训练数据的比重

    # Model 相关配置
    'max_len': 120,
    'train_batch_size': 8,
    'valid_batch_size': 4,
    'test_batch_size': 4,

    # 训练相关配置
    'lr': 2e-05,
    'epochs': 6,
}
```

2.2 清洗数据

由于原文中存在着一些诸如 `'-LRB-'` 与 `'-RRB-'` 的内容，因此我们需要先对输入进行清洗，并且去除一些像 `"!"` 与 `"."` 等可能会引起划分为两个句子的符号，否则会导致后续依存句法分析无法生成单棵句法树。

2.3 生成依存句法树

为了生成依存句法树，我选择了使用 `stanza` 库的 Dependency Parsing 模块，`stanza` 是由 Stanford University 开发的一个 NLP 库，其使用起来十分简单：

```
def depparse(documents):
    '''
    Input: a list of documents, each document is a string
    Output: a list of documents, each document is a stanza Document object
    '''
    nlp = stanza.Pipeline(
        lang='en', processors='tokenize,mwt,pos,lemma,depparse')
    in_docs = [stanza.Document([], text=d) for d in documents]
    out_docs = nlp(in_docs)
    return out_docs
```

输入一系列句子组成的 `documents`，经过处理之后，就会生成一系列的 stanza Document object，可以生成一系列类似于

```
id: 1   word: Nous      head id: 3      head: atteint   deprel: nsubj
id: 2   word: avons     head id: 3      head: atteint   deprel: aux:tense
id: 3   word: atteint   head id: 0      head: root      deprel: root
id: 4   word: la        head id: 5      head: fin       deprel: det
id: 5   word: fin       head id: 3      head: atteint   deprel: obj
id: 6   word: de        head id: 8      head: sentier   deprel: case
id: 7   word: le        head id: 8      head: sentier   deprel: det
id: 8   word: sentier   head id: 5      head: fin       deprel: nmod
id: 9   word: .         head id: 3      head: atteint   deprel: punct
```

的依存语法树。

2.4 寻找 target 根节点

在数据预处理中，会存在 target 不止一个 token 的情况，例如 `sauce on the pizza` 这个 target 就存在着 4 个 tokens。这时候我们需要确定其中一个 token，才能生成与其他 tokens 之间的距离 `dists`。

我确定 target 中的主 token 的策略是：寻找 target 中的 tokens 在依存句法树中离 root 节点最近的 token 作为 target root token。这是因为离 root 节点越近，就说明这个 token 在句子结构中就越为重要。

2.5 生成 token 权重

我们又使用 `build_graph(words)` 函数，去除边类型信息，为每棵句法树生成了形如 `[[3], [2], [1, 3], [2], [5], [4]]` 的图，并使用 BFS 算法找到 target 与其他 tokens 之间的最短距离 `dists`，其中 target 到 target 的距离也设为 `1`，以便后续生成权重 `weights`。

为了生成权重，我们有多种生成方式，基础的想法是距离越近的 token 的权重越大，距离越远的 token 的权重越小，甚至几乎等于 `0`。因此我使用了函数

$$h(x) = \frac{1}{x^2}$$

来生成最终的权重 `weights`，其优势在于距离为 `1` 的 token 的 weight 为 `1`，而距离越远的 token 的 weight 衰减得很快，变得十分接近 `0`，但是仍然不为 `0`，以保持一定的容错率。

2.6 BERT 输入分词

为了使用 BERT 模型，我们需要使用 `transformers` 库中的 `BertTokenizer` 对输入数据进行分词。关键代码如下：

```python
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
inputs = tokenizer.encode_plus(
    document,
```

```
    f"What is the sentiment class of {target} in the sentence ?",
    add_special_tokens=True,
    max_length=self.max_len,
    pad_to_max_length=True,
    return_token_type_ids=True
)

ids = inputs['input_ids']
mask = inputs['attention_mask']
token_type_ids = inputs["token_type_ids"]
```

可以看出分别会生成 `ids`、`mask` 以及 `token_type_ids` 三个变量，分别代表着 token 对应的 id，标识着有效输入部分的 mask 以及标记两个不同句子部分的 `token_type_ids`。

2.7 BERT 提示

可以看到，上文中的 `tokenizer.encode_plus()` 函数还包含了第二个句子作为提示参数：`"What is the sentiment class of {target} in the sentence ?"`,通过这种方式可以部分建模 target 对分类结果的影响。

这其实也相当于一个 BERT QUESTION ANSWER 模型，后一个句子就相当于我们的提问。借助 BERT 预训练模型的强大能力，在有了这么一个提示之后分类准确率就达到了 83.5% 左右了。

2.8 BERT 加权平均模型

使用 `transformers` 库的 `BertModel` 可以很简单地搭建出一个 BERT 预训练模型：

```
class TDBert(nn.Module):

    def __init__(self, num_labels=3):
        super(TDBert, self).__init__()
        self.bert = BertModel.from_pretrained('bert-base-uncased')
        self.classifier = nn.Linear(768, num_labels)


    def forward(self, input_ids, token_type_ids=None, attention_mask=None,
weights=None):
        last_hidden_state, pooled_output = self.bert(input_ids=input_ids,
attention_mask=attention_mask, token_type_ids=token_type_ids,
return_dict=False)
        weighted_output = torch.sum(last_hidden_state *
weights.unsqueeze(-1), dim=1) / torch.sum(weights, dim=1).unsqueeze(-1)
        logits = self.classifier(weighted_output)
        return logits
```

其中我们使用了 `BertModel` 与 `nn.Linear` 搭建了一个两层结构。但是实际上的 `forward()` 函数中我们还有一个加权平均的过程，这里应用到了我们前面预训练生成出来的 `weights`，具体为

```
weighted_output = torch.sum(last_hidden_state * weights.unsqueeze(-1),
dim=1) / torch.sum(weights, dim=1).unsqueeze(-1)
```

也即取出 `last_hidden_state`，其对应每一个 token 对应的 768 维的隐层表示，再乘上数据预处理时得到的权重 `weights` 进行加权平均，得到一个加权平均后的 768 维向量表示，再通过一个线性分类器 `nn.Linear` 分类得到最终的三个类别。

2.9 模型训练与评估

模型训练和评估使用的是 Adam 优化器与交叉熵损失函数，也即有

```
optimizer = torch.optim.Adam(model.parameters(), lr=config['lr'])
criterion = nn.CrossEntropyLoss()
```

然后使用 PyTorch 的梯度下降对模型进行训练和微调，其中一些关键参数为

三、 实验效果

经过数次的超参数调节，选定了 `max_len = 120, lr = 2e-05, epochs = 6` 等参数，并且 `train.txt` 中 90% 数据用作训练，10% 数据用作验证 (最后用了 99% 的数据重新训练模型)，最终得到了如下的实验效果：

训练集精度	验证集精度	测试集精度
0.972	0.847	0.850

表 1: 实验结果

在训练集上的精度为 0.972，在验证集上的精度为 0.847，最后提交文件得到的 `test.txt` 对应的精度为 0.850。

四、 遇到的问题与思考改进

4.1 超参数调节过于繁琐

在机器学习中，超参数调节是一个非常重要的环节，可以通过调节不同的超参数来优化模型性能。但是，在实际操作中，超参数调节往往非常繁琐，需要不断地试错，很容易浪费大量时间和计算资源。

由于我使用的是 Colab 的免费计算资源，连接容易中断，不适合进行大量的超参数调节。因此，我只能在 Colab 上进行少量的手动超参数调节。

如果是在本地训练，也许可以考虑使用自动化调参工具，例如 Grid Search、Random Search、Bayesian Optimization 等。

4.2 低级的代码错误导致网络无法正常训练

在一开始使用 BERT 模型的时候，在训练过程中，无论是在训练集还是验证集上，精度始终维持在 0.60 左右，无法进一步提升。

当时认为可能是训练函数 `train()` 出现了错误，一直在训练函数的身上寻找错误，但是没有找到问题所在。最后决定换一个网络 (transformers 自带的整句文本分类 BERT 网络) 测试一下训练函数 `train()` 是否能正常训练。结果表明训练函数 `train()` 是正常工作的，因此我就将目光放到了网络结构身上。

最后结果表明，出错的原因是我调用 BERT 模型时没有使用关键字参数，原来的写法是这样

```
self.bert(input_ids, attention_mask, token_type_ids, return_dict=False)
```

参数位置出现错配，只需要换成

```
self.bert(input_ids=input_ids, attention_mask=attention_mask,
token_type_ids=token_type_ids, return_dict=False)
```

也即关键字参数即可。

4.3 尝试 LSTM 方案

除了 BERT 网络，我也尝试使用了 MLP 模型，也即直接使用多层由 Relu 激活的线性层，辅以数据预处理得到的权重 weights，但是事实表明最终的结果并不好，最终精度只有 65% 左右。

在 MLP 模型失败之后，我也使用了 LSTM 模型，而其中的困难是如何将从数据预处理中使用依存句法树得到的权重 `weights` 应用到 LSTM 模型中。我采取了一个取巧的做法：**将权重乘以输入到 LSTM 模型的对应 token 的词向量**。

这个做法的原理在于词向量一般是需要归一化的，也即作为一个只有方向，而固定长度的单位向量。因此，我们可以让词向量乘以标量权重，得到一个新向量，这个新向量即有方向，又有大小，进而可以让 LSTM 网络学习到 target 相关的信息。

这样的 LSTM 模型性能比不上 BERT 模型，最终精度只有 74.5% 左右，但是我自己认为是一个蛮有趣的想法。

五、 参考文献

1. https://stanfordnlp.github.io/stanza/depparse.html
2. https://arxiv.org/pdf/2004.12362.pdf

3. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8864964
4. https://huggingface.co/docs/transformers/v4.29.1/en/model_doc/bert#overview