基于 BERT 的文本情感分类实践

姓　　名：李雅萌

学　　号：2201824

专 业 名 称：计算机科学与技术（硕）

2022 年 12 月

一 问题的提出

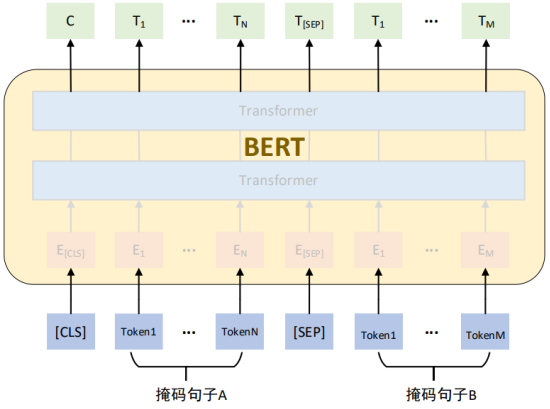
自然语言处理中的情感分析主要研究人类通过文字表达的情感，因此也称为文本情感分析。随着互联网的迅速发展，产生了各种各样的用户生成内容，其中很多内容包含着人们的喜怒哀惧等情感，对这些情感的准确分析有助于了解人们对某款产品的喜好，随时掌握舆情的发展。因此，情感分析成为目前自然语言处理技术的主要应用之一。

文本分类是最简单也是最基础的自然语言处理问题。即针对一段文本输入，输出该文本所属的类别，其中，类别是事先定义好的一个封闭的集合。文本分类具有众多的应用场景，如垃圾邮件过滤（将邮件分为垃圾和非垃圾两类)、新闻分类（将新闻分为政治、经济和体育等类别)等。

文本情感分类是情感分析的子任务，本实验使用 BERT 模型来进行中文文本情感分类任务。

二 相关理论和技术

BERT 是由多个 Transformer 堆叠而成的双向编码器表示模型，区别于以往利用单向语言模型或简单拼接两个单向语言模型进行预训练的方法，BERT 利用掩码语言模 型以及下一句子预测这两个无监督任务进行预训练，并得到深度双向表示。图为 BERT 进行预训练的示意图



BERT 的输入为句子中词的令牌以及两种特殊令牌的表示，两种特殊令牌分别为位于序列开端用于分类任务的[CLS]令牌，以及位于每个句子末尾用于区分两个句子的 [SEP]令牌。每个令牌的表示都由三种嵌入组成，分别是每个令牌的词嵌入、用于区分 不同句子的片段嵌入以及用于表示令牌位置的位置嵌入。输入表示经过深层双

向 Transformer 后的输出即为 BERT 的输出表示。其中，将[CLS]令牌对应的输出表示送到 额外输出层可用于处理句子级的分类任务，将其余令牌对应的输出表示送到额外的输 出层可用于处理令牌级的任务。 MLM 为 BERT 的预训练任务之一。MLM 先随机用[MASK]屏蔽 15%的输入令牌， 再将通过 BERT 训练得到的被屏蔽位置的输出表示输入到一个全连接层，并用 Softmax 计算每个令牌的概率，由此来预测出被屏蔽的令牌，MLM 通过上述过程训练深度双向表示。

三 实验及分析

1 模型准备

由于处理的是中文数据集，所以使用预处理模型 bert-base-chinese。模型是从 transformers 库中下载的。

数据集

本次大作业使用的数据集是微博评论对话数据集，原始数据源于新浪微博，由微热点大数据研究院提供。将微博按照其蕴含的情绪分为以下六个类别之一：积极、愤怒、悲伤、恐惧、惊奇和无情绪。数据集包括 8414 条对话数据。

实验环境

1. Python 3.8
2. anaconda 3
3. PyTorch 1.8
4. transformers 4.25.1

2 实验内容

使用预训练模型 BERT 进行中文文本情感分类，训练集中为多轮对话，每段对话有 n 个句子，每个句子都有对应的情绪标签，运行 main.py 训练模型，模型参数会保存在工程目录下，运行 text.py 加载模型进行测试。

参数设置

batch_size = 32, epoch=3, learning rate=1e-3

数据处理

对数据进行处理，将对话中的每句话和情绪进行匹配一同组成数据集

```python
def process_train(file_path, shuf=True):
    dataset = list()
    csv_reader = csv.reader(open(file_path,'r',encoding='utf-8'))
    for row in csv_reader:
        if row[2]:  # 标签存在
            strs = row[1].split('__eou__')
            labels = list(str(row[2]))
            for data in zip(strs, labels):
                dataset.append(data)

    data = pd.DataFrame(dataset, columns=['Text', 'Labels'])[1:]
    data['Labels'] = data['Labels'].astype(int)
    n_labels = len(set(data.Labels))

    cnt = Counter(list(data.Labels))

    label2idx = dict()
    idx2label = dict()
    for idx, label in enumerate(set(data.Labels)):
        label2idx[label] = idx
        idx2label[idx] = label
    for x in range(n_labels):
        assert label2idx[idx2label[x]] == x

    data['Labels'] = data.Labels.map(lambda x: label2idx[x])

    if shuf:
        data = shuffle(data)

    return data, n_labels, cnt
```

加载模型，对指定的模型使用 AutoModelForSequenceClassification.from_pretrained 进行加载，并设置相关参数

```python
def load_model_and_tokenizer(model_path, device, p_dropout=0.1, n_labels=None):
    logger.info('loading model and tokenizer from {} ...'.format(model_path))
    tokenizer = AutoTokenizer.from_pretrained(model_path)

    config = AutoConfig.from_pretrained(model_path)

    if hasattr(config, 'hidden_dropout_prob'):
        config.hidden_dropout_prob = p_dropout
    if hasattr(config, 'classifier_dropout'):
        config.classifier_dropout = p_dropout
    if hasattr(config, 'attention_probs_dropout_prob'):
        config.attention_probs_dropout_prob = p_dropout

    model = AutoModelForSequenceClassification.from_pretrained(model_path, config=config)

    if n_labels:
        hidden_size = model.config.hidden_size
        if hasattr(model.classifier, 'out_proj'):
            model.classifier.out_proj = nn.Linear(in_features=hidden_size, out_features=n_labels, bias=True)
        else:
            model.classifier = nn.Linear(in_features=hidden_size, out_features=n_labels, bias=True)
        model.config.num_labels = n_labels
    model.to(device)
    return model, tokenizer
```

对数据集进行划分和加载，返回数据迭代器：每次调用都会返回一个 batch_size 大小的数据

```python
def load_and_split_dataset(data, tokenizer, max_len, batch_size, test_size, with_label=True, shuf=True):
    if with_label == False:
        test_dataset = TextClassificationDataset(data["Text"], None, tokenizer, max_len)
        test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=shuf)
        return test_dataset, test_loader

    if test_size == 1.0:
        test_dataset = TextClassificationDataset(data["Text"], data["Labels"], tokenizer, max_len)
        test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=shuf)
        print(len(test_loader))
        return test_dataset, test_loader

    X_train, X_val, y_train, y_val = train_test_split(data["Text"], data["Labels"], test_size=test_size, random_state=1)
    train_dataset = TextClassificationDataset(X_train, y_train, tokenizer, max_len)
    val_dataset = TextClassificationDataset(X_val, y_val, tokenizer, max_len)
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=shuf)
    val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=shuf)
    return train_dataset, val_dataset, train_loader, val_loader
```

设置优化器，调整学习率的策略，以及训练过程

```python
optim = torch.optim.AdamW(model.parameters(), lr=2e-5)  # 1e-3
scheduler = get_linear_schedule_with_warmup(optim, num_warmup_steps=400, num_training_steps=len(train_loader) * n_epoch)
train(model, criterion, optim, scheduler, train_loader, val_loader, n_epoch=n_epoch, save_dir=save_dir, device=device)

del model, tokenizer, optim, scheduler, train_loader, val_loader
torch.cuda.empty_cache()
data = shuffle(data)
```

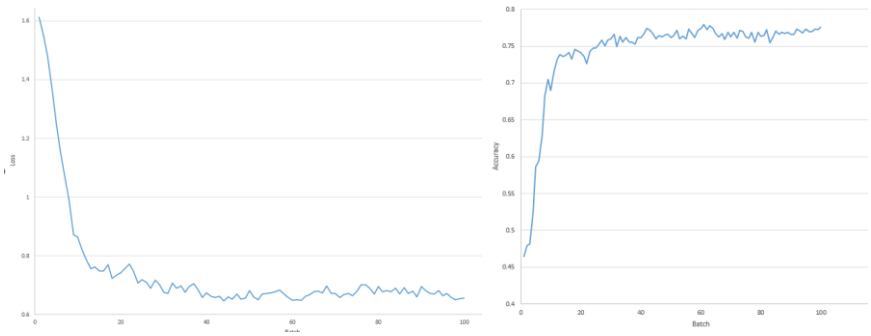
训练一轮次的代码，主要流程有：梯度清零，正向传播计算 loss，计算梯度并反向传播更新参数

```python
def train_epoch(model, criterion, optim, scheduler, train_loader, val_loader, epoch, train_log_interval=10, val_internal=50
    model.train()
    len_iter = len(train_loader)
    n_step = 0
    val_acces, val_fscores, val_losses = [], [], []
    for i, batch in enumerate(train_loader, start=1):
        optim.zero_grad()
        input_ids, attention_mask, labels = make_batch(batch, device)
        outputs = model(input_ids, attention_mask=attention_mask)
        loss = criterion(outputs.logits, labels)
        loss.backward()
        optim.step()

        n_step += 1
        if scheduler:
            scheduler.step()
        if i % train_log_interval == 0:
            logger.info("epoch: %d [%d/%d], loss: %.6f, lr: %.8f, steps: %d" %
                        (epoch, i, len_iter, loss.item(), optim.param_groups[0]["lr"], n_step + len_iter * (epoch-1)))
        if i % val_internal == 0:
            acc, score, loss = val_epoch(model, criterion, val_loader, save_dir, val_res, device)
            val_acces.append(acc)
            val_fscores.append(score)
            val_losses.append(loss)

    return val_acces, val_fscores, val_losses
```

3 实验结果与分析

```
23-01-11 20:51:20 | epoch: 3 [850/1036], loss: 0.069732, lr: 0.00000137, steps: 2922
23-01-11 20:51:22 | epoch: 3 [860/1036], loss: 0.123326, lr: 0.00000130, steps: 2932
23-01-11 20:51:27 | Valid | acc: 0.7719, score: 0.6815, global optim: 0.7194, loss: 0.8096
23-01-11 20:51:29 | epoch: 3 [870/1036], loss: 0.113600, lr: 0.00000123, steps: 2942
23-01-11 20:51:31 | epoch: 3 [880/1036], loss: 0.307530, lr: 0.00000115, steps: 2952
23-01-11 20:51:36 | Valid | acc: 0.7705, score: 0.6776, global optim: 0.7194, loss: 0.8054
23-01-11 20:51:38 | epoch: 3 [890/1036], loss: 0.111204, lr: 0.00000108, steps: 2962
23-01-11 20:51:40 | epoch: 3 [900/1036], loss: 0.055324, lr: 0.00000100, steps: 2972
23-01-11 20:51:45 | Valid | acc: 0.7700, score: 0.6795, global optim: 0.7194, loss: 0.8022
23-01-11 20:51:47 | epoch: 3 [910/1036], loss: 0.113322, lr: 0.00000093, steps: 2982
23-01-11 20:51:49 | epoch: 3 [920/1036], loss: 0.373921, lr: 0.00000086, steps: 2992
23-01-11 20:51:54 | Valid | acc: 0.7710, score: 0.6819, global optim: 0.7194, loss: 0.7929
23-01-11 20:51:56 | epoch: 3 [930/1036], loss: 0.209442, lr: 0.00000078, steps: 3002
23-01-11 20:51:58 | epoch: 3 [940/1036], loss: 0.071884, lr: 0.00000071, steps: 3012
23-01-11 20:52:03 | Valid | acc: 0.7719, score: 0.6842, global optim: 0.7194, loss: 0.7934
23-01-11 20:52:05 | epoch: 3 [950/1036], loss: 0.122206, lr: 0.00000064, steps: 3022
23-01-11 20:52:07 | epoch: 3 [960/1036], loss: 0.056978, lr: 0.00000056, steps: 3032
23-01-11 20:52:12 | Valid | acc: 0.7719, score: 0.6856, global optim: 0.7194, loss: 0.7944
23-01-11 20:52:14 | epoch: 3 [970/1036], loss: 0.211698, lr: 0.00000049, steps: 3042
23-01-11 20:52:16 | epoch: 3 [980/1036], loss: 0.046575, lr: 0.00000041, steps: 3052
23-01-11 20:52:21 | Valid | acc: 0.7727, score: 0.6837, global optim: 0.7194, loss: 0.7945
23-01-11 20:52:23 | epoch: 3 [990/1036], loss: 0.340912, lr: 0.00000034, steps: 3062
23-01-11 20:52:25 | epoch: 3 [1000/1036], loss: 0.102155, lr: 0.00000027, steps: 3072
23-01-11 20:52:30 | Valid | acc: 0.7721, score: 0.6845, global optim: 0.7194, loss: 0.7948
23-01-11 20:52:32 | epoch: 3 [1010/1036], loss: 0.155717, lr: 0.00000019, steps: 3082
23-01-11 20:52:34 | epoch: 3 [1020/1036], loss: 0.158884, lr: 0.00000012, steps: 3092
23-01-11 20:52:39 | Valid | acc: 0.7719, score: 0.6846, global optim: 0.7194, loss: 0.7963
23-01-11 20:52:41 | epoch: 3 [1030/1036], loss: 0.215822, lr: 0.00000004, steps: 3102
23-01-11 20:52:47 | Valid | acc: 0.7719, score: 0.6843, global optim: 0.7194, loss: 0.8043
```

按照 f-score 最高的策略保存训练的模型，最终在测试集中得到了 0.669 的 macro-f1 值，0.745 的准确率和 0.675 的召回率。



从结果可以看到 BERT 对于处理该类问题相较于其他模型而言是有效的，训练曲线如图所示，并且随着训练轮次的提升，模型效果也越来越好，但由于算力的限制，暂时运行了 3 个 epoch。我的代码暂时没有考虑，对于这类问题，其他论文的一些做法会考虑时间因素，因此之后改进的方向可以从时间因素出发进行。

BERT 模型在中文情感分类中的表现非常好，通过查阅资料了解到 BERT 的应用范围非常广泛，今后会尝试用 BERT 来处理其他任务。

四 总结与展望

本实践是在网上资料的参考下，边学习边实践，过程中也遇到了一些问题，通过查阅资料，同时在同学朋友的帮助下完成了本次实验。在学习和实践过程中，我更加深入地了解学习了自然语言处理和预训练模型，同时也认识到了自己的很多不足，比如代码能力较差等问题，今后在学习中一定多练习多思考，提升自己的实践能力。

最后，非常开心能够选修自然语言处理这门课程，感谢老师和师兄师姐们的指导，同时也感谢本次实践过程中同学朋友们对我的帮助。