10215501442的文本分类实验

训练数据划分

本次实验采用的是**固定划分**

根据CSDN上的这篇文章【深度学习】如何分配训练集、验证集、测试集比例_训练集测试集验证集比例-CSDN博客上 所说的内容

> 对于传统机器学习阶段（数据集在万这个数量级），一般分配比例为训练集和测试集的比例为7:3或是8:2。为了进一步降低信息泄露同时更准确的反应模型的效能，更为常见的划分比例是训练集、验证集、测试的比例为6：2：2。

因从，在本次实验中将**train_data.txt**的内容的 6/8 也就是**75%**分配为训练集，剩余的**25%**分配为验证集的内容。

而具体的分配是采取的随机分配，使用python程序读入**train_data.txt**的内容，统计一下各个label的个数，并对每个label随机分配75%至train.txt（训练集），剩余的25%分配至Validation.txt（验证集）。

divide.py:

```
import json
import random

# 初始化一个字典用于统计每个label的数量
label_counts = {str(i): 0 for i in range(10)}

# 读取a.txt文件，统计每个label的数量
with open('train_data.txt', 'r') as source_file:
    data = source_file.readlines()
for line in data:
    record = json.loads(line)
    label = str(record['label'])
    label_counts[label] += 1
# 打印每个label的数量
for label, count in label_counts.items():
    print(f"Label {label}: {count} 条数据")
# 随机分割数据并将其写入train.txt和check.txt
```

```
train_data = {}
check_data = {}
for label in label_counts.keys():
    data_for_label = [line for line in data if json.loads(line)['label'] ==
int(label)]
    random.shuffle(data_for_label)
    total_samples = len(data_for_label)
    train_samples = data_for_label[:int(total_samples * 0.75)]
    check_samples = data_for_label[int(total_samples * 0.75):]
    train_data[label] = train_samples
    check_data[label] = check_samples
# 写入train.txt
with open('train.txt', 'w') as train_file:
    for label in train_data:
        train_file.writelines(train_data[label])
# 写入check.txt
with open('Validation.txt', 'w') as check_file:
    for label in check_data:
        check_file.writelines(check_data[label])label in check_data:
```

数据以及Bert模型载入

本次实验的文本向量化选用的是Bert，Bert能够理解上下文中的单词，捕捉更全局的语义信息,而且可以用到Pytorch（在前期准备工作时为了装Pytorch我甚至重装了好几次anaconda）和GPU。（TF-IDF无法捕捉单词之间的语义关系，Word2Vec不适用于捕捉全局语境）

下面是数据以及模型载入的代码

```
print('预处理开始')
# 读取训练数据
data_path = r'train.txt的路径'
with open(data_path, "r") as file:
    train_data = file.read()
train_samples = [json.loads(line) for line in train_data.split('\n') if line.strip()]
print('训练数据载入成功')
# 读取验证数据
validation_data_path = r'Validation.txt的路径'
with open(validation_data_path, "r") as file:
    validation_data = file.read()
validation_samples = [json.loads(line) for line in validation_data.split('\n') if
line.strip()]
print('验证数据载入成功')
```

```
# 检查GPU是否可用
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("正在使用的device为:", device)

# 加载tokenizer
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
print('tokenizer已加载')

# 将模型加载到GPU（如果可用）或CPU
model = BertModel.from_pretrained("bert-base-uncased").to(device)
print('模型已加载至device')
```

文本向量化

以下是训练集的文本向量化代码

```
start_time = time.time() # 记录文本向量化的时间
train_vectors = []  # 用于存储训练集的文本向量
train_labels = []  # 用于存储训练集的标签

for sample in train_samples:
    raw_text = sample['raw']
    inputs = tokenizer(raw_text, return_tensors="pt", padding=True, truncation=True,
max_length=512)
    inputs = {key: value.to(device) for key, value in inputs.items()}
    with torch.no_grad():
        outputs = model(**inputs)
    hidden_states = outputs.last_hidden_state
    # 使用BERT的隐藏状态作为文本特征
    text_vector = hidden_states.mean(dim=1).squeeze()  # 使用均值池化
    label = sample['label']
    train_vectors.append(text_vector)
    train_labels.append(label)

train_vectors = torch.stack(train_vectors)
train_labels = torch.tensor(train_labels, dtype=torch.long).to(device)

# 验证数据的文本向量化，略

end_time = time.time()
print(f'文本向量化用时: {end_time - start_time} 秒')
```

在这里将训练集和验证集的文本进行了向量化，并使用了均值池化，均值池化可以确保最终的文本特征向量具有一致的维度，就以多次实验的观察来说，文本向量化（训练集和验证集）所需要的时间为3~4分钟之间，而在前面的数据以及Bert模型载入阶段也需要花费一分钟左右的时间。

```
预处理开始
训练数据载入成功
验证数据载入成功
正在使用的device为: cuda
tokenizer已加载
模型已加载至device
文本向量化用时：218.83988237380981 秒
```

测试集的文本向量化同理

训练分类器

MLP

```
# 超参数
hidden_layer = 1 # 当前隐藏层数 1
input_dim = 768  # BERT的隐藏状态大小
hidden_dim = 200 # 神经元数量
output_dim = 10  # 类别数
learning_rate = 0.1 # 学习率
epochs = 300     # 训练轮数
# 定义MLP模型
class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        #self.fc2 = nn.Linear(hidden_dim, hidden_dim)  # 第二个隐藏层
        #self.fc3 = nn.Linear(hidden_dim, hidden_dim)  # 第三个隐藏层
        self.fc4 = nn.Linear(hidden_dim, output_dim)  # 输出层

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        #x = torch.relu(self.fc2(x))
        #x = torch.relu(self.fc3(x))
        x = self.fc4(x)
        return x
# 打印当前超参数的代码，略，下面是训练模型以及进行验证的代码
start_time = time.time()
print('start training')
mlp_model = MLP(input_dim, hidden_dim, output_dim).to(device)
# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(mlp_model.parameters(), lr=learning_rate)
```

```
# 开始训练
for epoch in range(epochs):
 mlp_model.train()
 optimizer.zero_grad()
 outputs = mlp_model(train_vectors)
 loss = criterion(outputs, train_labels)
 loss.backward()
 optimizer.step()
end_time = time.time()
print(f'训练时间: {end_time - start_time} 秒')
```

如果要一次测试不同的超参数可以将上面超参数部分的内容更改为

```
epochs_s = [300,200,100]      # 训练轮数
将训练模型以及进行验证的代码放在这样的循环中
for epochs in epochs_s:
    .......
可设置多重循环
```

以多次实验的经验来看，模型训练时间一般很短，在一秒之内，因此多次训练不会花费大量时间。

以下的表格内的内容为验证集准确率，单位为%，加黑的为不同的超参数

神经元数量 100 隐藏层 1 **可以看出在训练轮数较低的情况下提高轮数可以提高验证集准确率**

轮数\学习率	**0.1**	**0.01**	**0.001**
10	31	10	9.7
20	54	9.2	7.5
30	66	11.8	10.45

轮数\学习率	**0.1**	**0.01**	**0.001**
50	82	12	8.9
75	81	21	10.6
100	84	45	11.6

神经元数量 100 学习率 0.1 **可以看出轮数不变的情况下层数少点更好**

轮数\隐藏层	2	3
100	75	33
200	76	54
300	86	61

学习率 0.1 隐藏层 1 **可能是神经元数量提高的不多，所以提升的不明显**

轮数\神经元	150	200
100	83	84
200	88	88
300	89	89

最终选择的超参数是

hidden_layer = 1 # 当前隐藏层数 3

input_dim = 768 # BERT的隐藏状态大小

hidden_dim = 200 # 神经元数量 output_dim = 10 # 类别数 learning_rate = 0.1 # 学习率 epochs = 300 # 训练轮数

验证集准确率为89~90%

测试

```
# 使用训练好的模型进行预测
mlp_model.eval()
with torch.no_grad():
    test_outputs = mlp_model(test_vectors)
    _, predicted_labels = torch.max(test_outputs, 1)
# 保存预测结果到result.txt
result_path = "result_MLP.txt"
with open(result_path, "w", encoding="utf-8") as result_file:
    result_file.write("id,pred\n")
    for i, label in enumerate(predicted_labels):
        result_file.write(f"{i},{label.item()}\n")
print("预测结果已保存到 result_MLP.txt")
#torch.save(mlp_model.state_dict(), 'MLP.pth')
```

逻辑回归

```
# 初始化逻辑回归模型
logistic_regression = LogisticRegression(max_iter=1000, random_state=42)
# 训练逻辑回归模型
logistic_regression.fit(train_texts, train_labels)
# 验证逻辑回归模型
predictions = logistic_regression.predict(validation_texts)
# 计算验证集准确率
accuracy = accuracy_score(validation_labels, predictions)
```

逻辑回归的验证集准确率为93.95%

SVM

```
svm_model = SVC(C=1.0, kernel='linear', gamma='scale')  # 超参数
svm_model.fit(train_texts, train_labels)
# 使用SVM模型进行验证
predictions = svm_model.predict(validation_texts)
# 计算验证集准确率
accuracy = accuracy_score(validation_labels, predictions)
print(f'验证集准确率: {accuracy}')
```

SVM的验证集准确率为93.9%

决策树

```
# 创建并训练决策树模型
decision_tree = DecisionTreeClassifier(max_depth=10)  # 设置决策树的超参数，可以根据需求调整
decision_tree.fit(loaded_train_texts, train_labels)
# 验证决策树模型
predictions = decision_tree.predict(loaded_validation_texts)
# 计算验证集准确率
accuracy = accuracy_score(validation_labels, predictions)
print(f'验证集准确率: {accuracy}')
```

决策树的验证集准确率为66.55

结果比较

```python
with open('result_svm.txt', 'r') as file1, open('result_Logistic.txt', 'r') as file2:
    lines1 = file1.readlines()
    lines2 = file2.readlines()

count = 0
total_lines = len(lines1) - 1
# 遍历每一行（从第二行开始），比较pred的值
for i in range(1, total_lines + 1):
    pred1 = float(lines1[i].split(',')[1])  # 假设pred在每行以逗号分隔
    pred2 = float(lines2[i].split(',')[1])
    if pred1 == pred2:
        count += 1
# 计算百分比
percentage = (count / total_lines) * 100
# 输出结果
print(f"相等的百分比：{percentage:.2f}%")
```

一个简单的比较结果之间相似度的程序，MLP，SVM，**逻辑回归互相之间的结果相似都在90%以上**，而**决策树与其他结果的比较相似度都在60~70%左右。**

BUG

有时载入tokenizer时，会有奇怪的报错，但是再次运行程序又没有了该问题，无法复现，非常奇怪，在五次六次运行中可能会有一次这样的报错。