第4次Python课程实验报告

姓名：李金旻 学号：2401210317 日期：2025.1.21

# 一、实验名称

第4次实验：BERT文本分类实验

# 二、实验介绍

该实验旨在利用BERT模型进行中文文本分类。实验任务是将新闻标题分类到10个类别中，包括财经、房产、股票、教育、科技、社会、时政、体育、游戏和娱乐。实验使用了从THUCNews中抽取的20万条新闻标题作为数据集，每条标题的文本长度在20到30个字之间。数据集被划分为训练集（18万条）、验证集（1万条）和测试集（1万条）。数据以字为单位输入到BERT模型中，以完成分类任务。

# 三、创新点

无

# 四、实验代码

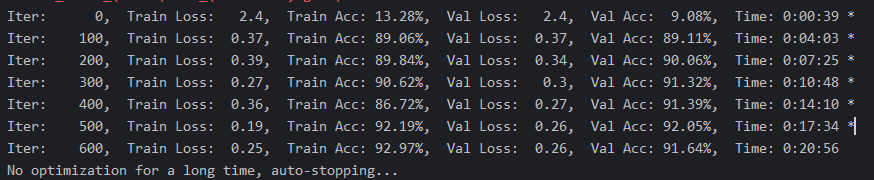
1.**run.py**

1. # coding: UTF-8
2. **import** time
3. **import** torch
4. **import** numpy as np
5. **from** train\_eval **import** train, init\_network
6. **from** importlib **import** import\_module
7. **import** argparse
8. **from** utils **import** build\_dataset, build\_iterator, get\_time\_dif
10. # 设置命令行参数解析
11. parser = argparse.ArgumentParser(description='Chinese Text Classification')
12. parser.add\_argument('--model', type=str, required=True, help='choose a model: Bert, ERNIE')
13. args = parser.parse\_args()
15. **if** \_\_name\_\_ == '\_\_main\_\_':
16. dataset = 'THUCNews'  # 使用的数据集名称
18. model\_name = args.model  # 从命令行参数获取模型名称，例如 'Bert'
19. x = import\_module('models.' + model\_name)  # 动态导入对应的模型模块
20. config = x.Config(dataset)  # 初始化模型的配置
22. # 设置随机种子以确保实验的可重复性
23. np.random.seed(1)
24. torch.manual\_seed(1)
25. torch.cuda.manual\_seed\_all(1)
26. torch.backends.cudnn.deterministic = True  # 确保每次运行的结果一致
28. # 检查CUDA是否可用，并打印当前设备信息
29. **print**("CUDA Available:", torch.cuda.is\_available())
30. **print**("Current Device:", torch.cuda.current\_device() **if** torch.cuda.is\_available() **else** "CPU")
32. start\_time = time.time()  # 记录开始时间
33. **print**("Loading data...")
34. # 构建训练集、验证集和测试集
35. train\_data, dev\_data, test\_data = build\_dataset(config)
36. # 构建数据迭代器
37. train\_iter = build\_iterator(train\_data, config)
38. dev\_iter = build\_iterator(dev\_data, config)
39. test\_iter = build\_iterator(test\_data, config)
40. time\_dif = get\_time\_dif(start\_time)  # 计算数据加载所用时间
41. **print**("Time usage:", time\_dif)
43. # 初始化模型并将其移动到指定的设备（GPU或CPU）
44. model = x.Model(config).to(config.device)
45. # 开始训练模型
46. train(config, model, train\_iter, dev\_iter, test\_iter)
47. **train\_eval.py**
48. # coding: UTF-8
49. **import** numpy as np
50. **import** torch
51. **import** torch.nn as nn
52. **import** torch.nn.functional as F
53. **from** sklearn **import** metrics
54. **import** time
55. **from** utils **import** get\_time\_dif
56. **from** pytorch\_pretrained.optimization **import** BertAdam
58. # 权重初始化函数，默认使用Xavier初始化
59. **def** init\_network(model, method='xavier', exclude='embedding', seed=123):
60. **for** name, w **in** model.named\_parameters():
61. **if** exclude **not** **in** name:  # 排除不需要初始化的层（如embedding层）
62. **if** len(w.size()) < 2:  # 跳过一维参数（如bias）
63. **continue**
64. **if** 'weight' **in** name:  # 初始化权重
65. **if** method == 'xavier':
66. nn.init.xavier\_normal\_(w)  # Xavier初始化
67. **elif** method == 'kaiming':
68. nn.init.kaiming\_normal\_(w)  # Kaiming初始化
69. **else**:
70. nn.init.normal\_(w)  # 普通正态分布初始化
71. **elif** 'bias' **in** name:  # 初始化偏置
72. nn.init.constant\_(w, 0)  # 偏置初始化为0
73. **else**:
74. **pass**
76. # 训练函数
77. **def** train(config, model, train\_iter, dev\_iter, test\_iter):
78. start\_time = time.time()  # 记录训练开始时间
79. model.train()  # 设置模型为训练模式
80. param\_optimizer = list(model.named\_parameters())  # 获取模型参数
81. no\_decay = ['bias', 'LayerNorm.bias', 'LayerNorm.weight']  # 不需要权重衰减的参数
82. optimizer\_grouped\_parameters = [
83. {'params': [p **for** n, p **in** param\_optimizer **if** **not** any(nd **in** n **for** nd **in** no\_decay)], 'weight\_decay': 0.01},  # 需要权重衰减的参数
84. {'params': [p **for** n, p **in** param\_optimizer **if** any(nd **in** n **for** nd **in** no\_decay)], 'weight\_decay': 0.0}  # 不需要权重衰减的参数
85. ]
86. # 使用BertAdam优化器
87. optimizer = BertAdam(optimizer\_grouped\_parameters,
88. lr=config.learning\_rate,
89. warmup=0.05,
90. t\_total=len(train\_iter) \* config.num\_epochs)
91. total\_batch = 0  # 记录当前训练的batch数
92. dev\_best\_loss = float('inf')  # 记录验证集上的最佳loss
93. last\_improve = 0  # 记录上次验证集loss下降的batch数
94. flag = False  # 标记是否长时间没有提升
95. model.train()  # 设置模型为训练模式
96. **for** epoch **in** range(config.num\_epochs):  # 遍历每个epoch
97. **print**('Epoch [{}/{}]'.format(epoch + 1, config.num\_epochs))
98. **for** i, (trains, labels) **in** enumerate(train\_iter):  # 遍历每个batch
99. # 将数据移动到指定设备（GPU或CPU）
100. trains = move\_to\_device(trains, config.device)
101. labels = labels.to(config.device)
102. outputs = model(trains)  # 前向传播
103. model.zero\_grad()  # 梯度清零
104. loss = F.cross\_entropy(outputs, labels)  # 计算损失
105. loss.backward()  # 反向传播
106. optimizer.step()  # 更新参数
107. **if** total\_batch % 20 == 0:  # 每20个batch输出一次训练和验证集的效果
108. true = labels.data.cpu()  # 获取真实标签
109. predic = torch.max(outputs.data, 1)[1].cpu()  # 获取预测标签
110. train\_acc = metrics.accuracy\_score(true, predic)  # 计算训练集准确率
111. dev\_acc, dev\_loss = evaluate(config, model, dev\_iter)  # 计算验证集准确率和损失
112. **if** dev\_loss < dev\_best\_loss:  # 如果验证集loss下降
113. dev\_best\_loss = dev\_loss  # 更新最佳loss
114. torch.save(model.state\_dict(), config.save\_path)  # 保存模型
115. improve = '\*'  # 标记有提升
116. last\_improve = total\_batch  # 记录提升的batch数
117. **else**:
118. improve = ''  # 无提升
119. time\_dif = get\_time\_dif(start\_time)  # 计算训练时间
120. msg = 'Iter: {0:>6},  Train Loss: {1:>5.2},  Train Acc: {2:>6.2%},  Val Loss: {3:>5.2},  Val Acc: {4:>6.2%},  Time: {5} {6}'
121. **print**(msg.format(total\_batch, loss.item(), train\_acc, dev\_loss, dev\_acc, time\_dif, improve))
122. model.train()  # 设置模型为训练模式
123. total\_batch += 1  # 更新batch数
124. **if** total\_batch - last\_improve > config.require\_improvement:  # 如果长时间没有提升
125. **print**("No optimization for a long time, auto-stopping...")  # 停止训练
126. flag = True
127. **break**
128. **if** flag:
129. **break**
130. test(config, model, test\_iter)  # 训练结束后进行测试
132. # 测试函数
133. **def** test(config, model, test\_iter):
134. model.load\_state\_dict(torch.load(config.save\_path))  # 加载最佳模型
135. model.eval()  # 设置模型为评估模式
136. start\_time = time.time()  # 记录测试开始时间
137. test\_acc, test\_loss, test\_report, test\_confusion = evaluate(config, model, test\_iter, test=True)  # 计算测试集效果
138. msg = 'Test Loss: {0:>5.2},  Test Acc: {1:>6.2%}'
139. **print**(msg.format(test\_loss, test\_acc))
140. **print**("Precision, Recall and F1-Score...")
141. **print**(test\_report)  # 打印分类报告
142. **print**("Confusion Matrix...")
143. **print**(test\_confusion)  # 打印混淆矩阵
144. time\_dif = get\_time\_dif(start\_time)  # 计算测试时间
145. **print**("Time usage:", time\_dif)
147. # 评估函数
148. **def** evaluate(config, model, data\_iter, test=False):
149. model.eval()  # 设置模型为评估模式
150. loss\_total = 0  # 记录总损失
151. predict\_all = np.array([], dtype=int)  # 记录所有预测结果
152. labels\_all = np.array([], dtype=int)  # 记录所有真实标签
153. with torch.no\_grad():  # 不计算梯度
154. **for** texts, labels **in** data\_iter:  # 遍历每个batch
155. texts = move\_to\_device(texts, config.device)  # 将数据移动到指定设备
156. labels = labels.to(config.device)
157. outputs = model(texts)  # 前向传播
158. loss = F.cross\_entropy(outputs, labels)  # 计算损失
159. loss\_total += loss  # 累加损失
160. labels = labels.data.cpu().numpy()  # 获取真实标签
161. predic = torch.max(outputs.data, 1)[1].cpu().numpy()  # 获取预测标签
162. labels\_all = np.append(labels\_all, labels)  # 记录真实标签
163. predict\_all = np.append(predict\_all, predic)  # 记录预测标签
165. acc = metrics.accuracy\_score(labels\_all, predict\_all)  # 计算准确率
166. **if** test:  # 如果是测试集，返回分类报告和混淆矩阵
167. report = metrics.classification\_report(labels\_all, predict\_all, target\_names=config.class\_list, digits=4)
168. confusion = metrics.confusion\_matrix(labels\_all, predict\_all)
169. **return** acc, loss\_total / len(data\_iter), report, confusion
170. **return** acc, loss\_total / len(data\_iter)  # 返回准确率和平均损失
172. # 将数据移动到指定设备的函数
173. **def** move\_to\_device(data, device):
174. """
175. 将输入数据（张量或元组）移动到指定设备。
176. """
177. **if** isinstance(data, tuple):  # 如果是元组，递归处理每个元素
178. **return** tuple(move\_to\_device(t, device) **for** t **in** data)
179. **elif** isinstance(data, torch.Tensor):  # 如果是张量，直接移动到设备
180. **return** data.to(device)
181. **else**:
182. **raise** TypeError(f"Unsupported data type: {type(data)}")  # 不支持的数据类型
183. **utils.py**
184. **from** datetime **import** timedelta
186. # 定义特殊符号
187. PAD, CLS = '[PAD]', '[CLS]'  # PAD用于填充，CLS是BERT中的分类符号
189. # 构建数据集的函数
190. **def** build\_dataset(config):
192. # 加载数据集并处理数据
193. **def** load\_dataset(path, pad\_size=32):
194. contents = []  # 用于存储处理后的数据
195. with open(path, 'r', encoding='UTF-8') as f:  # 打开文件
196. **for** line **in** tqdm(f):  # 使用tqdm显示进度条
197. lin = line.strip()  # 去除首尾空白字符
198. **if** **not** lin:  # 如果行为空，跳过
199. **continue**
200. content, label = lin.split('\t')  # 分割文本和标签
201. token = config.tokenizer.tokenize(content)  # 使用tokenizer对文本进行分词
202. token = [CLS] + token  # 在开头添加CLS符号
203. seq\_len = len(token)  # 记录序列长度
204. mask = []  # 初始化mask
205. token\_ids = config.tokenizer.convert\_tokens\_to\_ids(token)  # 将token转换为id
207. # 处理填充
208. **if** pad\_size:
209. **if** len(token) < pad\_size:  # 如果序列长度小于pad\_size，进行填充
210. mask = [1] \* len(token\_ids) + [0] \* (pad\_size - len(token))  # 生成mask
211. token\_ids += ([0] \* (pad\_size - len(token)))  # 填充token\_ids
212. **else**:  # 如果序列长度大于pad\_size，进行截断
213. mask = [1] \* pad\_size
214. token\_ids = token\_ids[:pad\_size]
215. seq\_len = pad\_size
216. contents.append((token\_ids, int(label), seq\_len, mask))  # 将处理后的数据添加到contents中
217. **return** contents
219. # 加载训练集、验证集和测试集
220. train = load\_dataset(config.train\_path, config.pad\_size)
221. dev = load\_dataset(config.dev\_path, config.pad\_size)
222. test = load\_dataset(config.test\_path, config.pad\_size)
223. **return** train, dev, test
225. # 数据集迭代器类
226. **class** DatasetIterater(object):
227. **def** \_\_init\_\_(self, batches, batch\_size, device):
228. self.batch\_size = batch\_size  # 每个batch的大小
229. self.batches = batches  # 所有数据
230. self.n\_batches = len(batches) // batch\_size  # 计算batch的数量
231. self.residue = False  # 记录batch数量是否为整数
232. **if** len(batches) % self.n\_batches != 0:  # 如果不能整除，说明有剩余数据
233. self.residue = True
234. self.index = 0  # 当前batch的索引
235. self.device = device  # 设备（GPU或CPU）
237. # 将数据转换为张量并移动到指定设备
238. **def** \_to\_tensor(self, datas):
239. x = torch.LongTensor([\_[0] **for** \_ **in** datas]).to(self.device)  # 输入数据
240. y = torch.LongTensor([\_[1] **for** \_ **in** datas]).to(self.device)  # 标签
242. # pad前的长度(超过pad\_size的设为pad\_size)
243. seq\_len = torch.LongTensor([\_[2] **for** \_ **in** datas]).to(self.device)  # 序列长度
244. mask = torch.LongTensor([\_[3] **for** \_ **in** datas]).to(self.device)  # mask
245. **return** (x, seq\_len, mask), y
247. # 获取下一个batch
248. **def** \_\_next\_\_(self):
249. **if** self.residue **and** self.index == self.n\_batches:  # 处理剩余的batch
250. batches = self.batches[self.index \* self.batch\_size: len(self.batches)]
251. self.index += 1
252. batches = self.\_to\_tensor(batches)
253. **return** batches
255. **elif** self.index >= self.n\_batches:  # 如果所有batch都已遍历，重置索引并抛出StopIteration
256. self.index = 0
257. **raise** StopIteration
258. **else**:  # 正常获取下一个batch
259. batches = self.batches[self.index \* self.batch\_size: (self.index + 1) \* self.batch\_size]
260. self.index += 1
261. batches = self.\_to\_tensor(batches)
262. **return** batches
264. # 返回迭代器自身
265. **def** \_\_iter\_\_(self):
266. **return** self
268. # 返回batch的数量
269. **def** \_\_len\_\_(self):
270. **if** self.residue:
271. **return** self.n\_batches + 1  # 如果有剩余数据，batch数量加1
272. **else**:
273. **return** self.n\_batches
275. # 构建数据迭代器
276. **def** build\_iterator(dataset, config):
277. iter = DatasetIterater(dataset, config.batch\_size, config.device)
278. **return** iter
280. # 获取已使用时间的函数
281. **def** get\_time\_dif(start\_time):
282. """获取已使用时间"""
283. end\_time = time.time()
284. time\_dif = end\_time - start\_time
285. **return** timedelta(seconds=int(round(time\_dif)))  # 返回时间差
286. **bert.py**
287. # coding: UTF-8
288. **import** torch
289. **import** torch.nn as nn
290. # from pytorch\_pretrained\_bert import BertModel, BertTokenizer  # 旧版BERT库
291. **from** pytorch\_pretrained **import** BertModel, BertTokenizer  # 导入BERT模型和Tokenizer
293. # 配置类，用于定义模型的超参数和路径
294. **class** Config(object):
296. """配置参数"""
297. **def** \_\_init\_\_(self, dataset):
298. self.model\_name = 'bert'  # 模型名称
299. self.train\_path = dataset + '/data/train.txt'  # 训练集路径
300. self.dev\_path = dataset + '/data/dev.txt'  # 验证集路径
301. self.test\_path = dataset + '/data/test.txt'  # 测试集路径
302. # 读取类别名单
303. self.class\_list = [x.strip() **for** x **in** open(
304. dataset + '/data/class.txt').readlines()]
305. # 模型保存路径
306. self.save\_path = dataset + '/saved\_dict/' + self.model\_name + '.ckpt'
307. # 设备（GPU或CPU）
308. self.device = torch.device('cuda' **if** torch.cuda.is\_available() **else** 'cpu')
310. self.require\_improvement = 1000  # 若超过1000个batch效果没有提升，则提前结束训练
311. self.num\_classes = len(self.class\_list)  # 类别数量
312. self.num\_epochs = 3  # 训练的epoch数
313. self.batch\_size = 128  # 每个mini-batch的大小
314. self.pad\_size = 32  # 每句话处理成的长度（短填长切）
315. self.learning\_rate = 5e-5  # 学习率
316. self.bert\_path = './bert\_pretrain'  # 预训练BERT模型的路径
317. # 加载BERT的Tokenizer
318. self.tokenizer = BertTokenizer.from\_pretrained(self.bert\_path)
319. self.hidden\_size = 768  # BERT模型的隐藏层大小
321. # 定义模型类
322. **class** Model(nn.Module):
324. **def** \_\_init\_\_(self, config):
325. super(Model, self).\_\_init\_\_()
326. # 加载预训练的BERT模型
327. self.bert = BertModel.from\_pretrained(config.bert\_path)
328. # 设置BERT模型的参数为可训练
329. **for** param **in** self.bert.parameters():
330. param.requires\_grad = True
331. # 定义一个全连接层，将BERT的输出映射到类别数量
332. self.fc = nn.Linear(config.hidden\_size, config.num\_classes)
333. # 定义前向传播过程
334. **def** forward(self, x):
335. context = x[0]  # 输入的句子（token IDs）
336. mask = x[2]  # 对padding部分进行mask，padding部分用0表示
337. # 通过BERT模型获取输出
338. \_, pooled = self.bert(context, attention\_mask=mask, output\_all\_encoded\_layers=False)
339. # 将BERT的输出通过全连接层进行分类
340. out = self.fc(pooled)
341. **return** out

# 五、运行结果

1.BERT测试

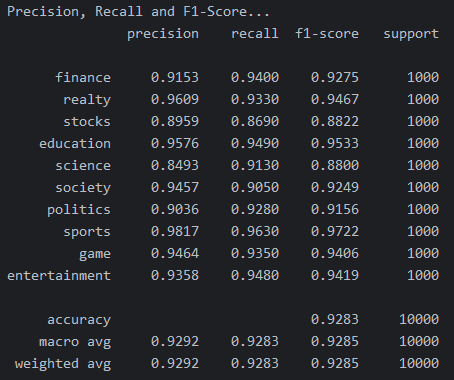
训练过程，每100个Batch测试一次



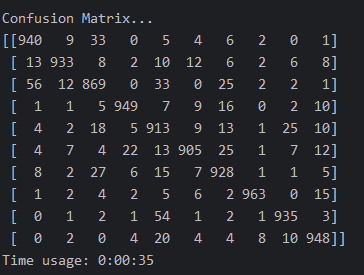
运行结果，测试准确率92.83%



准确率、召回率及f1分数

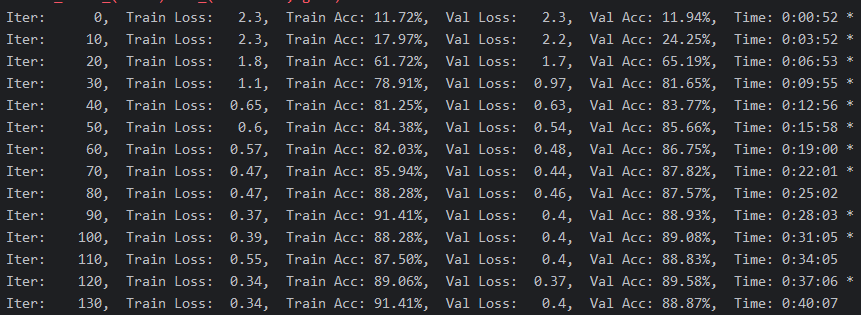


混淆矩阵



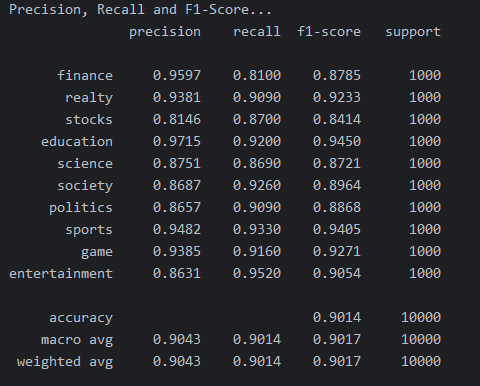
2.BERT\_CNN测试

训练过程，每10个Batch测试一次

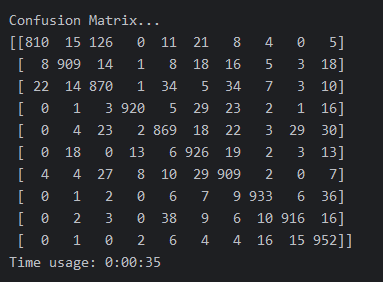


运行结果，测试准确率90.14%

准确率、召回率及f1分数



混淆矩阵



# 六、参考

项目来源：https://github.com/649453952/Bert-Chinese-Text-Classification-Pytorch