作家风格识别

1.实验介绍

1.1实验背景

作家风格是作家在作品中表现出来的独特的审美风貌。通过分析作品的写作风格来识别作者这一研究有很多应用，比如可以帮助人们鉴定某些存在争议的文学作品的作者、判断文章是否剽窃他人作品等。作者识别其实就是一个文本分类的过程，文本分类就是在给定的分类体系下，根据文本的内容自动地确定文本所关联的类别。写作风格学就是通过统计的方法来分析作者的写作风格，作者的写作风格是其在语言文字表达活动中的个人言语特征，是人格在语言活动中的某种体现。

1.2 实验要求

a）建立深度神经网络模型，对一段文本信息进行检测识别出该文本对应的作者。

b）绘制深度神经网络模型图、绘制并分析学习曲线。

c）用准确率等指标对模型进行评估。

1.3 实验环境

可以使用基于 Python 分词库进行文本分词处理，使用 Numpy 库进行相关数值运算，使用 Keras 等框架建立深度学习模型等。

2.实验内容与实验过程

2.1 介绍数据集

该数据集包含了 8438 个经典中国文学作品片段，对应文件分别以作家姓名的首字母大写命名。
数据集中的作品片段分别取自 5 位作家的经典作品，分别是：

序号	中文名	英文名	文本片段个数
1	鲁迅	LX	1500 条
2	莫言	MY	2219 条
3	钱钟书	QZS	1419 条
4	王小波	WXB	1300 条
5	张爱玲	ZAL	2000 条

- 其中截取的片段长度在 100~200 个中文字符不等
- 数据集路径为 `dataset/` 以作者名字首字母缩写命名

读取数据集，保存在字典中

```
dataset = {}
path = "dataset/"
files= os.listdir(path)
for file in files:
    if not os.path.isdir(file) and not file[0] == '.': # 跳过隐藏文件和文件夹
        f = open(path+"/"+file, 'r',  encoding='UTF-8'); # 打开文件
        for line in f.readlines():
            dataset[line] = file[:-4]
```

数据集总共有 8438 个文本片段，现在展示其中的 6 个片段及其作者。

```
name_zh = {'LX': '鲁迅', 'MY':'莫言' , 'QZS':'钱钟书' ,'WXB':'王小波' ,'ZAL':'张爱玲'}
for (k,v) in  list(dataset.items())[:6]:
    print(k,'---',name_zh[v])
```

```
几个少年辛苦奔走了十多年，暗地里一颗弹丸要了他的性命；几个少年一击不中，在监牢里身受一个多月的苦刑；几个少年怀着远志，忽然踪影全无，连尸首也不知那里去了。—“他们都在社会的冷笑恶骂迫害倾陷里过了一生；现在他们的坟墓也早在忘却里渐渐平塌下去了。
 --- 鲁迅
是的，你不解的。他一面点灯，一面冷静地说，“你的和我交往，我想，还正因为那时的哭哩。你不知道，这祖母，是我父亲的继母；他的生母，他三岁时候就死去了。他想着，默默地喝酒，吃完了一个熏鱼头。“那些往事，我原是不知道的。
 --- 鲁迅
这声音虽然极低，却很耳熟。看时又全没有人。站起来向外一望，那孔乙己便在柜台下对了门槛坐着。他脸上黑而且瘦，已经不成样子；穿一件破夹袄，盘着两腿，下面垫一个蒲包，用草绳在肩上挂住；见了我，又说道，“温一碗酒。
 --- 鲁迅
“阿，—纬甫，是你么？我万想不到会在这里遇见你。“阿阿，是你？我就邀他同坐，但他似乎略略踌躇之后，方才坐下来。我起先很以为奇，接着便有些悲伤，而且不快了。细看他相貌，也还是乱蓬蓬的须发；苍白的长方脸，然而衰瘦了。
 --- 鲁迅
我惶恐着，站起来说。“那么，我对你说。迅哥儿，你阔了，搬动又笨重，你还要什么这些破烂木器，让我拿去罢。我们小户人家，用得着。“我并没有阔哩。“阿呀呀，你放了道台⑼了，还说不阔？你现在有三房姨太太；出门便是八抬的大轿，还说不阔？
 --- 鲁迅
老三多两个孩子上学，老五也说他多用了公众的钱，气不过……。“这真是愈加闹不清了！月生失望似的说。“所以看见你们弟兄，沛君，我真是‘五体投地’。是的，我敢说，这决不是当面恭维的话。沛君不开口，望见听差的送进一件公文来，便迎上去接在手里。
 --- 鲁迅
```

2.2 数据集预处理

在做文本挖掘的时候，首先要做的预处理就是分词。英文单词天然有空格隔开容易按照空格分词，但是也有时候需要把多个单词做为一个分词，比如一些名词如 "New York"，需要做为一个词看待。而中文由于没有空格，分词就是一个需要专门去解决的问题了。这里我们使用 jieba 包进行分词，使用**精确模式**、**全模式**和**搜索引擎模式**进行分词对比。

```
# 精确模式分词
titles = [".".join(jb.cut(t, cut_all=False)) for t,_ in dataset.items()]
print("精确模式分词结果:\n",titles[0])
```

```
# 全模式分词
titles = [".".join(jb.cut(t, cut_all=True)) for t,_ in dataset.items()]
print("全模式分词结果:\n",titles[0])
```

```
# 搜索引擎模式分词
titles = [".".join(jb.cut_for_search(t)) for t,_ in dataset.items()]
print("搜索引擎模式分词结果:\n",titles[0])
```

程序结果如下：

```
精确模式分词结果:
 几个.少年.辛苦.奔走.了.十多年.,.暗地里.一颗.弹丸.要.了.他.的.性命.;.几个.少年.一击.不.中.,.在.监牢.里.身受.一个多月.的.苦刑.;.几个.少年.怀着.远志.,.忽然.踪影.全无.,.连.尸首.也.不知.那里.去.了.。.—.-.-.“.他们.都.在.社会.的.冷笑.恶骂.迫害.倾陷.里.过.了.一生.;.现在.他们.的.坟墓.也.早.在.忘却.里.渐渐.平.塌下去.了.。.
```

```
全模式分词结果：
 几个.少年.辛苦.奔走.了.十多.十多年.多年.，.暗地.暗地里.地里.一颗.弹丸.要.了.他.的.性命.；.几个.少年.一.击.不中.，.在.监牢.里.身受.一个.一个多.一个多月.多月.的.苦刑.；.几个.少年.怀着.远志.，.忽然.踪影.全.无.，.连.尸首.也.不知.那里.去.了.。—“.他们.都.在.社会.的.冷笑.恶骂.迫害.倾陷.里.过.了.一生.；.现在.他们.的.坟墓.也.早在.忘却.里.渐渐.平.塌下.塌下去.下去.了.。.

搜索引擎模式分词结果：
 几个.少年.辛苦.奔走.了.十多.多年.十多年.，.暗地.地里.暗地里.一颗.弹丸.要.了.他.的.性命.；.几个.少年.一击.不.中.，.在.监牢.里.身受.一个.多月.一个多.一个多月.的.苦刑.；.几个.少年.怀着.远志.，.忽然.踪影.全无.，.连.尸首.也.不知.那里.去.了.。.—.—.“.他们.都.在.社会.的.冷笑.恶骂.迫害.倾陷.里.过.了.一生.；.现在.他们.的.坟墓.也.早.在.忘却.里.渐渐.平.塌下.下去.塌下去.了.。.
```

使用 TF-IDF 算法统计各个作品的关键词频率

TF-IDF（term frequency–inverse document frequency，词频-逆向文件频率）是一种用于信息检索与文本挖掘的常用加权技术。

- TF-IDF是一种统计方法，用以评估一字词对于一个文件集或一个语料库中的其中一份文件的重要程度。字词的重要性随着它在文件中出现的次数成正比增加，但同时会随着它在语料库中出现的频率成反比下降。
- TF-IDF的主要思想是：如果某个单词在一篇文章中出现的频率TF高，并且在其他文章中很少出现，则认为此词或者短语具有很好的类别区分能力，适合用来分类。这里我们使用 jieba 中的默认语料库来进行关键词抽取，并展示每位作者前 5 个关键词。

```python
# 将片段进行词频统计
str_full = {}
str_full['LX'] = ""
str_full['MY'] = ""
str_full['QZS'] = ""
str_full['WXB'] = ""
str_full['ZAL'] = ""

for (k,v) in dataset.items():
    str_full[v] += k

for (k,v) in str_full.items():
    print(k,":")
    for x, w in jb.analyse.extract_tags(v, topK=5, withWeight=True):
        print('%s %s' % (x, w))
```

程序结果如下：

```
LX :
阿Q 0.05379690966906414
没有 0.03501956188388567
一个 0.02659384736489112
知道 0.026370791166196325
什么 0.026117200927953624
MY :
西门 0.04035127611822447
父亲 0.03577176072663162
我们 0.02835442224012238
金龙 0.0274694159504008
一个 0.024059865345607147
QZS :
鸿渐 0.22516872869267315
辛楣 0.12453008658571695
小姐 0.06799326435687081
孙小姐 0.06114419277994029
柔嘉 0.05635906861892125
WXB :
李靖 0.05500282755382402
海鹰 0.048308857103309115
但是 0.03985236017697917
后来 0.028965598554340735
假如 0.026102821217101606
ZAL :
太太 0.05627531927572857
霓喜 0.02639100786806079
一个 0.02441434637731472
没有 0.022718951368287554
自己 0.019893114499557583
```

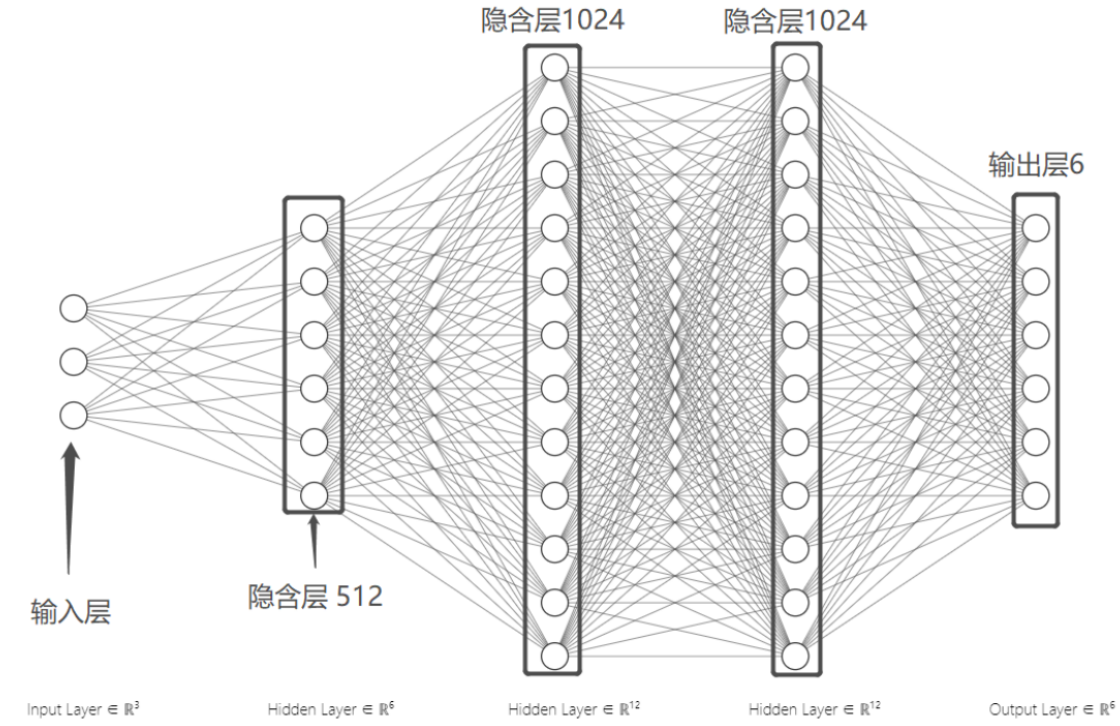
分析以上结果，我们选用精确模式分词，理由如下：

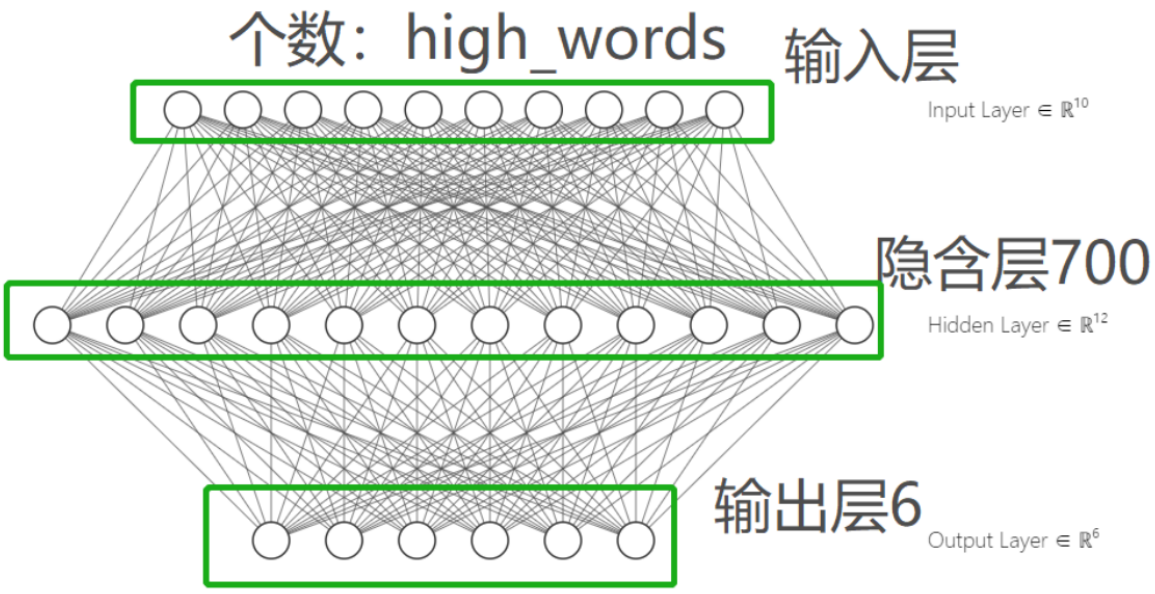
- 可以准确提取分词
- 其中重复的字词较少，如“十多年”、“暗地里”等，在后续选择特征时可以极大化覆盖的范围，避免出现特征的集中化；
- 充分利用特征空间，能够增大差异，便于训练；
- 分词相对较少，有利于降低内存空间压力，提升训练性能。

同时，我们也发现对于一些常用词，如“一个”等常用语，很容易出现多个作者共同使用且被提取为特征的情况，因此我们将注重于降低这些词语作为作家特征的权重，以提升模型的泛化性能。

2.3 采用 Pytorch 建立一个简单的深度神经网络模型

神经网络结构如图所示：





通过 Pytorch 构建深度学习模型的步骤如下：

- 准备数据，构建Dataset
- 定义模型、损失函数和优化器
- 迭代训练，进行反向传播和梯度下降
- 模型保存和测试评估

```
def load_data(path):
    """
    读取数据和标签
    :param path:数据集文件夹路径
    :return:返回读取的片段和对应的标签
    """
    sentences = [] # 片段
```

```
    target = [] # 作者

    # 定义lebel到数字的映射关系
    labels = {'LX': 0, 'MY': 1, 'QZS': 2, 'WXB': 3, 'ZAL': 4}

    files = os.listdir(path)
    for file in files:
        if not os.path.isdir(file):
            f = open(path + "/" + file, 'r', encoding='UTF-8');  # 打开文件
            for index,line in enumerate(f.readlines()):
                sentences.append(line)
                target.append(labels[file[:-4]])

    return list(zip(sentences, target))
```

- 创建词汇表

```
# 定义Field
TEXT  = Field(sequential=True, tokenize=lambda x: jb.lcut(x), lower=True,
use_vocab=True)
LABEL = Field(sequential=False, use_vocab=False)
FIELDS = [('text', TEXT), ('category', LABEL)]

# 读取数据，是由tuple组成的列表形式
mydata = load_data(path)

# 使用Example构建Dataset
examples = list(map(lambda x: Example.fromlist(list(x), fields=FIELDS), mydata))
dataset = Dataset(examples, fields=FIELDS)
# 构建中文词汇表
TEXT.build_vocab(dataset)
```

- 创建数据集迭代器

```
# 切分数据集
train, val = dataset.split(split_ratio=0.7)

# 生成可迭代的mini-batch
train_iter, val_iter = BucketIterator.splits(
    (train,val), # 数据集
    batch_sizes=(8, 8),
    device=-1, # 如果使用gpu，此处将-1更换为GPU的编号
    sort_key=lambda x: len(x.text),
    sort_within_batch=False,
    repeat=False
)
```

- 定义损失函数和优化器

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters())
```

- 迭代训练，进行反向传播和梯度下降

- 加载模型并进行预测

2.4 实验分析

在上述流程中，我们对于各种相互可替代方法的选用如下：

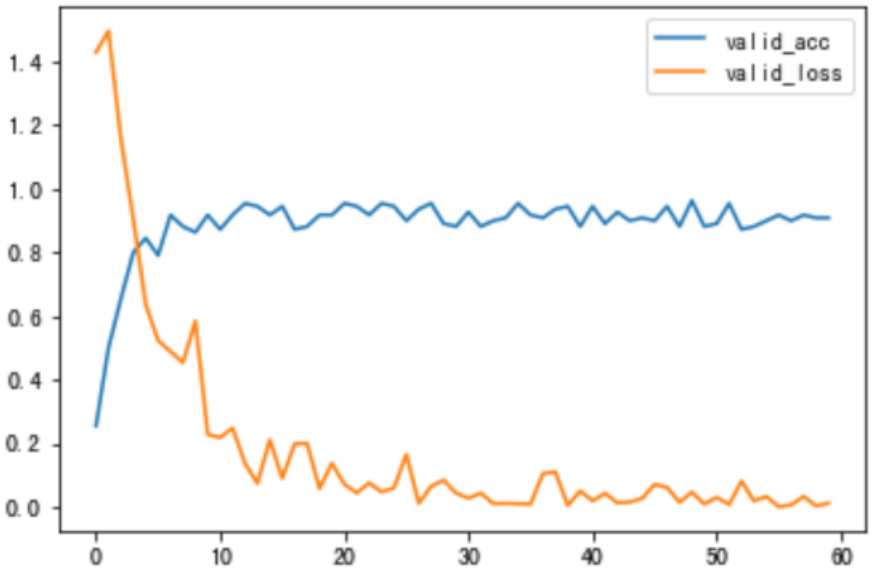
- 特征词汇选择：选用500个高频词，防止选用过多出现趋于同一化，也防止特征较少难以识别的问题。同时去除相同的高频词；
- 分词模式：采用精确分词模式
- 损失函数选择：`nn.CrossEntropyLoss()`
- 优化器选择：`torch.optim.Adam(model.parameters())`

3. 实验过程、结果与分析

题目要求根据一段中文文本（100~200 个中文字符），预测这段文本的作者。

3.1 创建并训练模型

我们首先进行初次实验，选定训练轮数为60，结果如下：



测试详情

测试点	状态	时长	结果
	✓	5s	测试完成 一共50个文本，预测正确48个

确定

我们发现在该模型中，还有一定的提升空间。

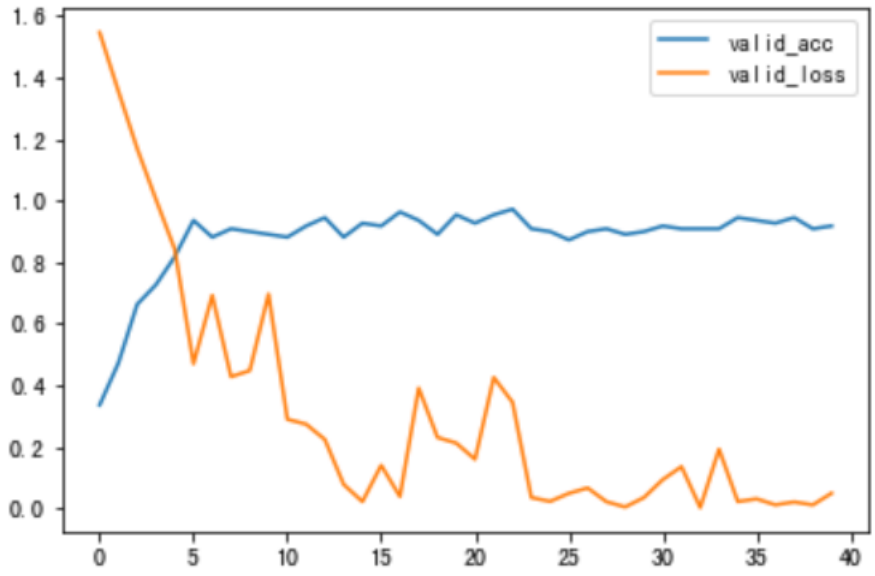
3.2 模型优化与重复训练

分析程序训练过程的图像可见，训练过程中很早就出现了较大的准确性，后续提升并不明显，推测为过拟合的影响。相应的调整其他超参数，最终选定：

- 验证集占比：25%
- 训练轮次：40
- 分词个数：每个作家500，去除重复分词
- batch size：16

因此，我们减小模型训练轮数，优化相关参数后重新测试，结果如下：

```
epoch:32 | valid_acc:0.9091
epoch:33 | valid_acc:0.9091
epoch:34 | valid_acc:0.9455
epoch:35 | valid_acc:0.9364
epoch:36 | valid_acc:0.9273
epoch:37 | valid_acc:0.9455
epoch:38 | valid_acc:0.9091
epoch:39 | valid_acc:0.9182
best accuracy:1.0000
```



测试详情

测试点	状态	时长	结果
	✓	4s	测试完成 一共50个文本，预测正确49个

确定

这一次优化后训练集上的准确率没有下降，但测试结果的准确率提升。同时训练过程中`loss`和`acc`与之前的结果类似，交叉验证了这一现象。同时由于降低了训练轮数，整体模型的训练时间也有明显的缩短。

总之，最终实验结果较好，能够满足模型的预测功能。

4. 心得与体会

这次实验中实现了一个简单的深度神经网络实现作家风格识别的程序，整体的过程思路还是比较清晰明了，分为选词特征提取、深度神经网络训练等，其中机器学习模型构建为重要的参数调整内容。

在整体实验过程中发现，尽管训练轮数已经下降了20次，但整体训练模型的时间消耗仍然比较大，考虑可以进一步使用GPU模型训练等方式降低时间消耗。同时这一问题也是机器学习中的特点，训练集的数据量必须足够大才能够获得满意的结果，但往往会使得学习时间更长，这也是当下机器学习模型的重要挑战之一。

在训练过程中，我们对于其中的超参数进行了一定的优化。在深度神经网络中过拟合现象是最为容易产生的问题之一，我们结合训练过程进行问题的发现并处理，通过调整训练轮数等超参数，使得整个模型及时停止，降低过拟合的问题，实现了在良好结果上的进一步提高，实验取得了满意的结果。

5. 附录：模型训练代码：

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

number_to_author = ['LX', 'MY', 'QZS', 'WXB', 'ZAL']  # 作家集合
author_number = len(number_to_author)  # 作家数量
author_to_number = {author: i for i, author in enumerate(number_to_author)}  # 建立作家数字映射，从0开始

# 读入数据集
data_begin = []  # 初始数据集合
path = 'dataset/'  # 数据路径
# path = 'test_data/test_case1_data/'  # 数据路径
for file in os.listdir(path):
    if not os.path.isdir(file) and not file[0] == '.':  # 跳过隐藏文件和文件夹
        with open(os.path.join(path, file), 'r',  encoding='UTF-8') as f:  # 打开文件
            for line in f.readlines():
                data_begin.append((line, author_to_number[file[:-4]]))

# 将片段组合在一起后进行词频统计
fragment = ['' for _ in range(author_number)]
for sentence, label in data_begin:
    fragment[label] += sentence   # 每个作家的所有作品组合到一起

# 词频特征统计，取出各个作家前 500 的词
high_words = set()
for label, text in enumerate(fragment):  # 提取每个作家频率前500的词汇，不返回关键词权重值
    for word in jb.analyse.extract_tags(text, topK=500, withWeight=False):
        if word in high_words:
            high_words.remove(word)
        else:
            high_words.add(word) # 将高频词汇存入

number_to_word = list(high_words)
word_number = len(number_to_word)  # 所有高频词汇的个数
word_to_number = {word: i for i, word in enumerate(number_to_word)} # 建立高频词汇字典，一一对应

features = torch.zeros((len(data_begin), word_number))
```

```
labels = torch.zeros(len(data_begin))
for i, (sentence, author_belong) in enumerate(data_begin):
    feature = torch.zeros(word_number, dtype=torch.float)
    for word in jb.lcut(sentence):  # jb.lcut 直接生成的就是一个list，尝试
jb.lcut_for_search()搜索引擎模式和jb.lcut精确分词模式
        if word in high_words:
            feature[word_to_number[word]] += 1
    if feature.sum():
        feature /= feature.sum()
        features[i] = feature
        labels[i] = author_belong
    else:
        labels[i] = 5  # 表示识别不了作者

dataset = data.TensorDataset(features, labels)

# 划分数据集
valid_weight = 0.25  # 25%验证集
train_size = int((1 - valid_weight) * len(dataset))  # 训练集大小
valid_size = len(dataset) - train_size  # 验证集大小
train_dataset, test_dataset = torch.utils.data.random_split(dataset,
[train_size, valid_size])
# 创建一个 DataLoader 对象
train_loader = data.DataLoader(train_dataset, batch_size=16, shuffle=True)  #
batch_size=16
valid_loader = data.DataLoader(test_dataset, batch_size=1000, shuffle=True)  #
batch_size=1000

# 设定模型参数，使用ReLU作为激活函数，简单顺序连接模型
model = nn.Sequential(
    nn.Linear(word_number, 700),  # 一个隐含层神经网络，尝试（700）
    nn.ReLU(),  # 激活函数尝试ReLU，
    nn.Linear(700, 6),
      # 最后一个隐含层不需要激活函数
).to(device)

epochs = 40  # 设定训练轮次
loss_fn = nn.CrossEntropyLoss()  # 定义损失函数（尝试nn.CrossEntropyLoss()和
nn.NLLLoss(),二者多用于多分类任务）
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)  # 定义优化器（adam），初
始学习率为1e-4
best_acc = 0   # 优化器尝试RMSProp（）、Adam（）、Adamax（）
history_acc = []
history_loss = []
best_model = model.cpu().state_dict().copy()  # 最优模型

for epoch in range(epochs):  # 开始训练
    for step, (word_x, word_y) in enumerate(train_loader):
        word_x = word_x.to(device)  # 传递数据
        word_y = word_y.to(device)
        out = model(word_x)
        loss = loss_fn(out, word_y.long())  # 计算损失
        optimizer.zero_grad()
        loss.backward()  # 反向传播
        optimizer.step()
```

```
        train_acc = np.mean((torch.argmax(out, 1) == word_y).cpu().numpy())

        with torch.no_grad():  # 上下文管理器，被包裹语句不会被track
            for word_x, word_y in valid_loader:
                word_x = word_x.to(device)
                word_y = word_y.to(device)
                out = model(word_x)
                valid_acc = np.mean((torch.argmax(out, 1) ==
word_y).cpu().numpy())  # 准确率求平均
        if valid_acc > best_acc:  # 记录最佳模型
            best_acc = valid_acc
            best_model = model.cpu().state_dict().copy()
    print('epoch:%d | valid_acc:%.4f' % (epoch, valid_acc))   # 展示训练过程
    history_acc.append(valid_acc)
    history_loss.append(loss)

print('best accuracy:%.4f' % (best_acc, ))
torch.save({
    'word2int': word_to_number,
    'int2author': number_to_author,
    'model': best_model,
}, 'results/temp.pth')  # 保存模型

plt.plot(history_acc,label = 'valid_acc')
plt.plot(history_loss,label = 'valid_loss')
plt.legend()
plt.show()
```

在模型预测部分，我们在预测函数前进行相关自定义模型的代码如下，这将作为测试部分的标准：

```
import torch
import jieba as jb

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# 加载模型(请加载你认为的最佳模型)
model_path = 'results/temp.pth'  # 模型路径
checkpoint = torch.load(model_path)

# 提取词典和作家信息
word_to_number = checkpoint['word2int']
number_to_author = checkpoint['int2author']
word_number = len(word_to_number)

# 创建模型实例
model = torch.nn.Sequential(
    torch.nn.Linear(word_number, 700),
    torch.nn.ReLU(),
    torch.nn.Linear(700, 6)
).to(device)

model.load_state_dict(checkpoint['model'])
model.eval()  # 设置为评估模式
```