案例四：酒店评论情感分析

1.案例目的

• 掌握情感分析任务的建模方法；

• 掌握BiLstm模型的搭建；

• 掌握torchtext工具的使用；

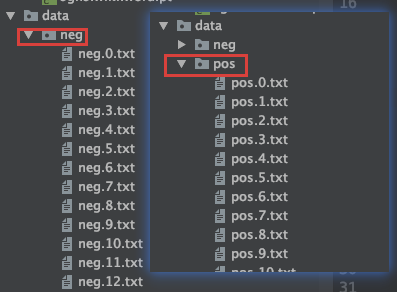
• 掌握使用flask部署模型；

2.案例内容

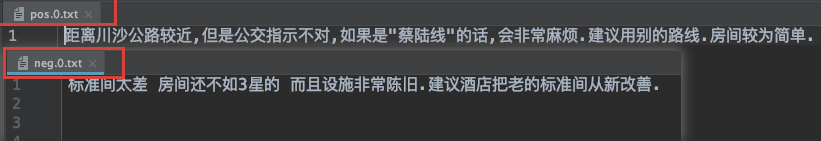
在线下消费时,大家已经习惯了在美团、口碑、携程等网站上查看该商户的评价，以确定自己是否在这家店进行消费。每一家商户在这些APP上都有大量的评论信息，有的是正面的评论信息，有的是负面的评论信息。一款智能的APP能自动的对用户的评价进行分类。

本案例使用携程APP的酒店评价数据，结合自然语言处理技术和机器学习方法，建立自然语言情感分析模型。该模型能自动的对用户评价的情感进行分析判别，判断该评论是对商户的正面评论还是负面评论。

本案例提供的数据文件包含了neg和pos文件夹，各含有3000个评论文本如下图所示:



文本内容如下图所示:



其中neg代表“消极”，pos代表“积极”。

3.案例知识点

• 文本预处理；

• BiLstm建模；

• torchtext工具使用；

• flask框架部署模型并开放接口；

• 调用接口进行情感分类；

4.案例时长

共4学时，学时分配如下：

• 文本预处理（1学时）；

• 模型训练和评估（2.5学时）

• 模型部署和预测（0.5学时）

5.案例实验环境

• Windows 10 操作系统；

• Pycharm

• Python3.6

• pytorch 1.6.0

• flask 1.1.2

• torchtext 0.7.0

• requests 2.24.0

6.案例分析

1）创建项目并拷贝数据文件

2）文本预处理

3）模型训练和评估

4）flask部署和调用

7.案例实验过程

7.1创建项目并拷贝数据文件

1）创建项目

使用Pycharm工具，创建一个 Project，项目名称为“Sentiment-Analysis”。

2）创建“data”目录

在Sentiment-Analysis项目中目录上点右键，在弹出的菜单中选择“NewDirectory” ，创建一个新目录，如图4所示。

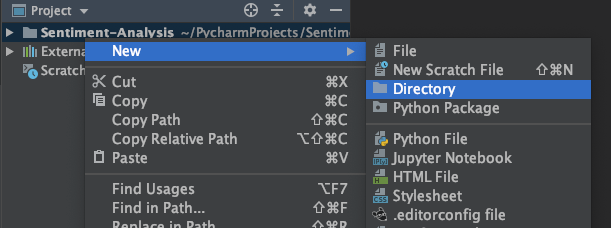


图4 创建目录菜单

在“New Directory”向导对话框，如图5所示，在对话框中需要进行以下操作：

1）创建目录“data”；

4）点击“OK”按钮。

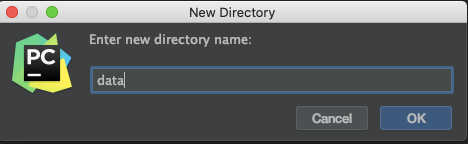


图5 创建目录向导

3）创建其他目录

按照同样的方法创建目录“model”、“model\_storage”，最终结果如图6所示：

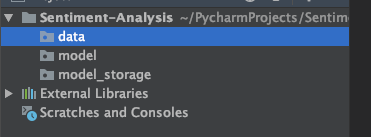


图6 创建目录结果

4）复制文件

将实案例提供的数据文件复制到“data”目录下，结构如图7所示：

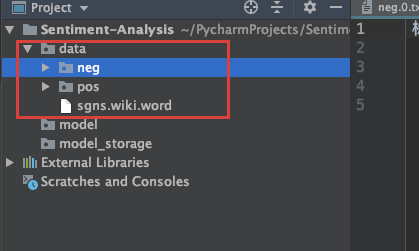


图7 复制数据文件结果

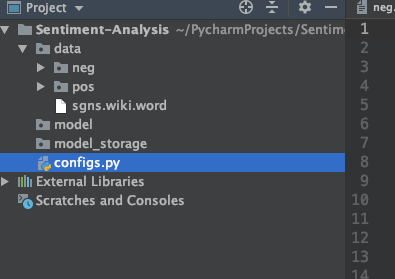
其中各个文件的作用为：

• neg：负面评论文本,共3000个文本文件

• pos：正面评论文本,共3000个文本文件

• sgns.wiki.word：维基百科预训练的词向量

将案例提供的脚本文件configs.py复制到“Sentiment-Analysis”根目录下，结构如下图所示：



该文件作用是配置案例中的文件路径、模型训练参数进行配置，内容如下：

import torch

class BasicConfigs():

#数据存放参数

neg='data/neg'#负样本目录

pos='data/pos'#正样本目录

data\_path='data'#分割后数据存放目录

text\_vocab\_path='model\_storage/text.vocab'#文本词典存放目录

label\_vocab\_path ='model\_storage/label.vocab'#标签词典存放目录

stop\_word\_path='data/stopword.txt'#停用词文件路径

#词向量参数

embedding\_loc = 'data/sgns.wiki.word'#词向量文件路径

#模型训练参数

device = 'cuda' if torch.cuda.is\_available() else 'cpu'#设备

lr = 0.001#学习率

dropout\_rate = 0.5#随机失活比例

train\_embedding = True#是否训练词嵌入向量

batch\_size = 64#批次大小

alpha=0.0001#L2惩罚项系数

#textcnn配置参数

kernel\_sizes = [3, 4, 5] # 3个 conv1d的size

num\_channels = [100, 100, 100]#卷积核个数

# bilstm配置参数

num\_hiddens = 100#lstm神经元数

num\_layers = 1#lstm层数

save\_model\_dir={#模型保存路径

'textcnn':'model\_storage/model\_cnn.pt',

'birnn':'model\_storage/model\_rnn.pt'

}

各参数含义在文件中已做解释。

7.2文本预处理

由于我们拿到的正负样本数据都是分散在3000个文本文件中，因此我们需要对数据进行读取、整理、分割、分词并创建词典。因此，我们需要新建一个脚本文件process\_data，并在文件中编写如下几个函数：

• load\_data\_to\_csv：该函数负责读取正负样本数据并进行整理、分割和保存；

• clearTxt：该函数负责清洗文本，取出文本中的特殊符号和标点；

• my\_cut：该函数负责分词；

• prepare\_vocab：该函数负责构建词典和训练集/测试集迭代数据；

• transform\_data：该函数负责在文本预测是完成文本分词、词典转换的预处理工作；

首先，在Sentiment-Analysis项目中目录上点右键，在弹出的菜单中选择“NewPython File” ，创建一个新py文件，如图8所示：

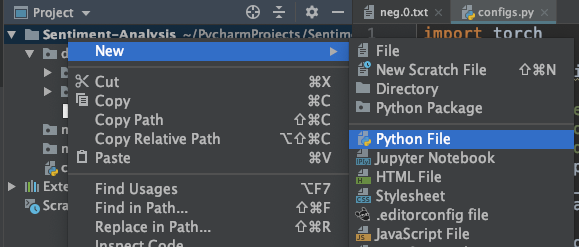


图8 新建py文件菜单

在“New Python File”向导对话框，如图9所示，在对话框中需要进行以下操作：

1）填写Name“process\_data”；

4）点击“OK”按钮。

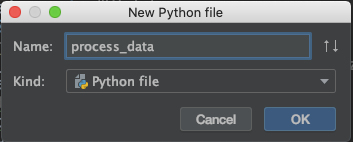


图9 新建py文件向导

在process\_data.py文件中，编写我们需要的函数。

1）导入使用的包。

这里我们导入了以下工具:

• pandas：用来构造DataFrame并存储csv文件;

• train\_test\_split：用来分割数据集；

• BasicConfigs：导入上一步中的配置参数

• glob：用来遍历文件夹中的所有文件

• jieba：用来分词

• pickle：用来读取和存储二进制（词典）文件

• torchtext：用来构建训练字段、词向量、词典和批次迭代器

• re：正则匹配，负责清洗文本

【代码1】data.py

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from configs import BasicConfigs

from glob import glob

import jieba

import pickle

from torchtext.vocab import Vectors

from torchtext.data import Field, TabularDataset, BucketIterator

import re

bc = BasicConfigs()#读取配置参数

2）编写load\_data\_to\_csv函数代码

该函数无需接收参数，需要读取的文件目录和保存文件路径都已经在BasicConfigs中进行了配置，这里已经在上一小步中读取，存储在对象bc中。

在load\_data\_to\_csv函数中，我们主要进行了以下工作：

• 遍历neg和pos目录，读取正负样本文本数据；

• 将所有数据打乱顺序并分割为训练集、验证集和测试集

• 将训练集、验证集和测试集分别保存到对应的csv文件

下面，我们首先实现这个函数，在process\_data.py文件中输入如下代码：

【代码2】process\_data.py

def load\_data\_to\_csv():

#遍历文件夹 读取数据

contents=[]

for file in glob(bc.neg+"/\*.txt"):#读取neg目录中的所有文件

with open(file,'r',encoding='utf-8') as f:

content=''.join([line.strip() for line in f.readlines()])#读取文本

contents.append([content,'neg'])#添加标签

for file in glob(bc.pos+"/\*.txt"):#读取pos目录中的所有文件

with open(file,'r',encoding='utf-8') as f:

content = ''.join([line.strip() for line in f.readlines()])

contents.append([content,'pos'])

#打乱顺序并存储到train.csv,test.csv,val.csv

#封装df

df=pd.DataFrame(contents,columns=['text','label'])

train,test=train\_test\_split(df,test\_size=0.1,random\_state=12)#数据分割

train,val=train\_test\_split(train,test\_size=0.2,random\_state=12)#训练集再分割

train.to\_csv(bc.data\_path+'/train.csv',index=False)#保存数据

val.to\_csv(bc.data\_path + '/val.csv', index=False)

test.to\_csv(bc.data\_path + '/test.csv', index=False)

print("文件处理完毕")

3）编写clearTxt函数和my\_cut函数代码

这里我们主要是实现文本清洗和分词功能，其中：

• clearTxt函数：负责清洗文本，将文本中的特殊符号和标点符号去除；

• my\_cut函数：首先调用clearTxt函数对文本进行清洗，然后使用jieba进行分词，并返回分此后的词列表；

这两个函数的参数都只有一个：line 一行文本

在process\_data.py文件中输入如下代码：

【代码3】process\_data.py

# 文本清洗

def clearTxt(line):

if line != '':

line = line.strip()

# 去除文本中的中文符号和英文符号

line = re.sub("[\s+\.\!\/\_,$%^\*(+\"\'；：“”．]+|[+——！，。？?、~@#￥%……&\*（）]+", "", line)

return line

def my\_cut(line):

line=clearTxt(line)#清洗

return jieba.lcut(line)#分词并返回中

4）编写prepare\_vocab函数代码

prepare\_vocab函数接收一个布尔型参数is\_train：

• 值为True：表示是训练阶段，训练阶段需要读取文本分词并构建词表和标签映射表，然后读取词向量，最后构建训练和验证集的批次迭代管理器；

• 值为False：表示是预测阶段，此时只需要读取训练时保存的词表和标签映射表并返回；

prepare\_vocab函数中我们完成了比较多的文本数据处理工作，具体如下：

• 定义训练和预测需要使用的数据字段TEXT和LABEL；

• 读取训练集和验证集文件并；

• 读取词向量文件

• 创建词表和标签映射表

• 创建训练集和验证集的批次迭代器

• 读取词表和标签映射表

首先，我们先定义构件计算图需要使用的字段TEXT和LABEL，并和数据中的列名进行映射。在process\_data.py文件中输入如下代码：

【代码4】process\_data.py

def prepare\_vocab(is\_train=True):

# 定义Field

TEXT = Field(tokenize=my\_cut)

LABEL = Field(eos\_token=None, pad\_token=None, unk\_token=None)

# 定义字段与FIELD之间读配对

fields = [('text', TEXT), ('label', LABEL)]

接着，我们编写在is\_train=True时的流程，即在训练阶段的流程，主要工作如下：

• 读取训练和验证数据

• 读取词向量

• 创建词表和标签映射表

• 创建训练集和验证集的批次迭代器

• 保存词表和标签映射表，以备预测是进行读取使用

• 返回字段、迭代器和词表形状

我们继续在process\_data.py文件中编写该函数，代码如下：

【代码5】process\_data.py

if is\_train:

# 读取数据

train\_data, val\_data = TabularDataset.splits(path=bc.data\_path, train='train.csv',

validation='val.csv',

format='csv',

fields=fields,

skip\_header=True)

# 构建从本地加载的词向量

vectors = Vectors(name=bc.embedding\_loc)

# 词表和标签映射表

TEXT.build\_vocab(train\_data, val\_data, vectors=vectors)

LABEL.build\_vocab(train\_data, val\_data)

#创建训练集和验证集的批次迭代器

train\_iter = BucketIterator(train\_data,

batch\_size=bc.batch\_size,

sort\_key=lambda x: len(x.text),

sort\_within\_batch=True,

shuffle=True)

val\_iter = BucketIterator(val\_data,

batch\_size=bc.batch\_size,

sort\_key=lambda x: len(x.text),

sort\_within\_batch=True,

shuffle=True)

#保存词表和标签映射表

with open(bc.text\_vocab\_path, 'wb')as f:

pickle.dump(TEXT.vocab, f)

with open(bc.label\_vocab\_path, 'wb')as f:

pickle.dump(LABEL.vocab, f)

vocab\_size = TEXT.vocab.vectors.shape

return TEXT,LABEL,train\_iter,val\_iter,vocab\_size

最后，我们编写在is\_train=False时的流程，即在预测阶段的流程，主要工作如下：

• 读取词表和标签映射表

• 将读取的词表和标签映射表分别配置到对应的字段

• 返回字段和词表形状

我们继续在process\_data.py文件中编写该函数，代码如下：

【代码6】process\_data.py

else:

# 加载词典

with open(bc.text\_vocab\_path, 'rb')as f:

TEXT.vocab = pickle.load(f)#词典配置到字段

#加载标签词表

with open(bc.label\_vocab\_path, 'rb')as f:

LABEL.vocab = pickle.load(f)#配置

vocab\_size = TEXT.vocab.vectors.shape#得到词表形状

return TEXT,LABEL,vocab\_size

5）编写预测时处理数据的函数transform\_data

在预测阶段，我们需要对文本进行跟训练时处理方法一样的处理，包括：

• 文本清洗和分词

• 使用词表对文本进行id和向量映射

因此，该函数接收如下三个参数：

• record：文本数据或者（文本数据，标签）

• TEXT：已经加载词表的TEXT字段，用来对分词结果进行映射

• LABEL：已经加载标签词表的LABEL字段，用来对标签进行映射

我们继续在process\_data.py文件中编写该函数，代码如下：

【代码7】process\_data.py

def transform\_data(record, TEXT, LABEL):

if not isinstance(record, dict):

raise ValueError('Make sure data is dict')

tokens = my\_cut(record['text'])#文本清洗和分词

res = []

for token in tokens:#遍历每一个词，进行映射

res.append(TEXT.vocab.stoi[token])

data = torch.tensor(res).unsqueeze(1)

if 'label' in list(record):#如果有标签

#对标签进行映射

label = torch.tensor(LABEL.vocab.stoi[record['label']])

else:

label = None

return data, label

6）编写main代码对数据进行整理和分割并运行

编写代码，调用load\_data\_to\_csv函数实现对数据进行整理、分割和保存：

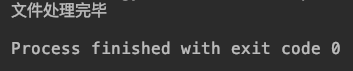
我们继续在process\_data.py文件中编写该函数，代码如下：

**【代码8】process\_data.py**

if \_\_name\_\_ == '\_\_main\_\_':

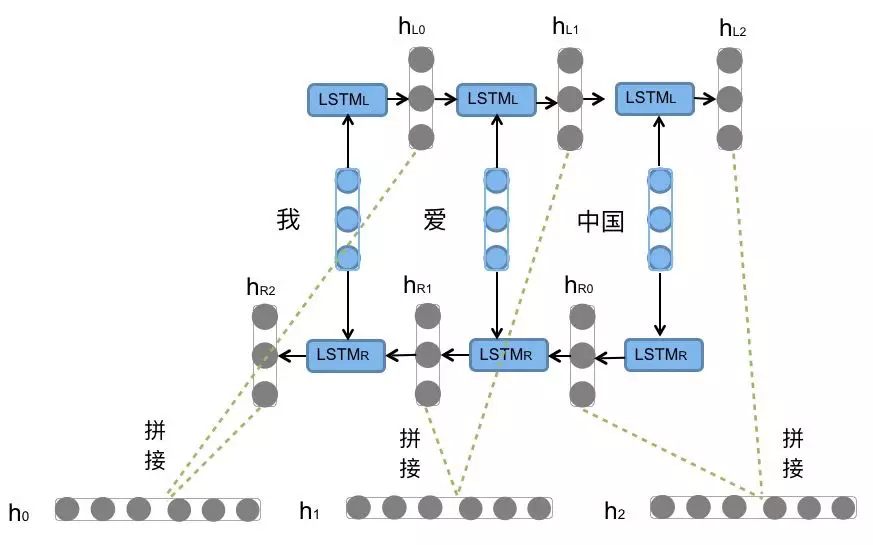
load\_data\_to\_csv()

编写完成后，运行process\_data.py文件，结果如下图所示：



7.3模型训练和评估

本次案例我们使用的模型为BiLstm模型，前向的LSTM与后向的LSTM结合成BiLSTM。比如，我们对“我爱中国”这句话进行编码，模型如图所示：



前向的



依次输入“我”，“爱”，“中国”得到三个向量{



,



,



}。后向的



依次输入“中国”，“爱”，“我”得到三个向量{



,



,



}。最后将前向和后向的隐向量进行拼接得到{[



,



], [



,



], [



,



]}，即{



,



,



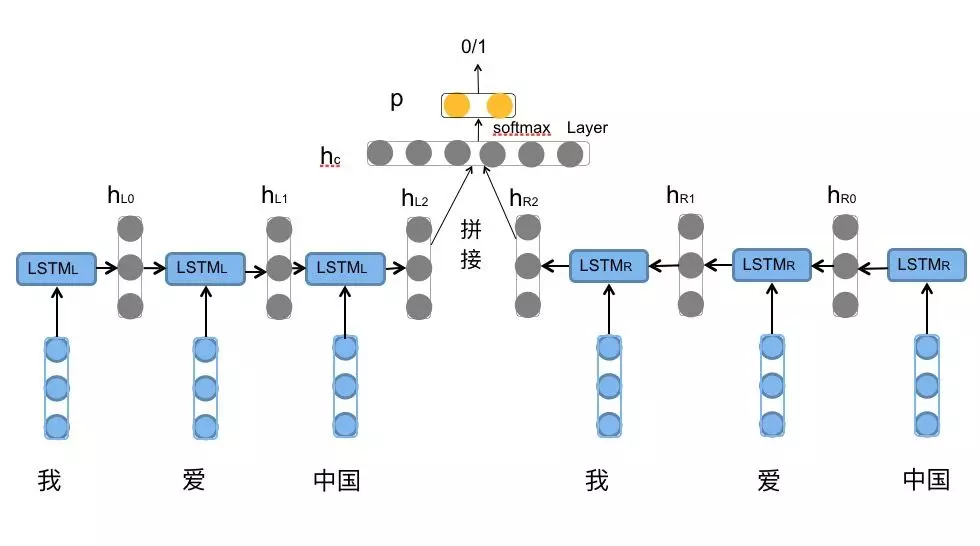
}。对于情感分类任务来说，我们采用的句子的表示往往是[



,



]。因为其包含了前向与后向的所有信息，如图所示：



BiLstm层包含两个部分：

• Embedding层：Embedding层的主要作用是将每个token（字）嵌入成为固定长度的词向量，然后送入BiLstm层进行计算

• BiLstm层：BiLstm层作用是利用双向LSTM的机制，提取文本的上下文语义特征，并为每个token输出一个状态向量；

下面我们来创建实现BiLstm层的类BiLSTM

在项目Sentiment-Analysis的model目录下新建Python File文件“birnn”。

1）导入需要使用的包并加载配置参数。

【代码9】birnn.py

import torch

from torch import nn

from configs import BasicConfigs

bc = BasicConfigs()#加载配置参数

2）编写类BiRNN

我们首先实现BiRNN类的\_\_init\_\_初始化方法，该方法接收一下参数：

• TEXT：文本字段，该字段中已经携带了词典和对应的词向量

• vocab\_size：词典形状

• num\_hiddens：lstm神经元数

• num\_layers：lstm层数

在该方法中，我们主要实现以下几个层：

• embedding：词向量嵌入层，并加载TEXT中携带的词向量权重

• encoder：多层双向LSTM层

• decoder：全连接映射层，负责映射分类

我们再birnn.py文件中编写如下代码：

【代码10】birnn.py

class BiRNN(nn.Module):

def \_\_init\_\_(self,TEXT,vocab\_size, num\_hiddens=bc.num\_hiddens, num\_layers=bc.num\_layers):

super(BiRNN, self).\_\_init\_\_()

#嵌入层

self.embedding = nn.Embedding(vocab\_size[0], vocab\_size[1])

#加载词向量权重

self.embedding.weight.data.copy\_(TEXT.vocab.vectors)

# bidirectional设为True即得到双向循环神经网络

self.encoder = nn.LSTM(input\_size=vocab\_size[1],

hidden\_size=num\_hiddens,

num\_layers=num\_layers,

bidirectional=True)

# 初始时间步和最终时间步的隐藏状态作为全连接层输入

self.decoder = nn.Linear(4\*num\_hiddens, 2)

3）编写前向计算方法forward

该方法实现biLstm的前向计算。代码如下：

【代码11】】birnn.py

def forward(self, inputs):

# 再提取词特征，输出形状为(词数, 批量大小, 词向量维度)

embeddings = self.embedding(inputs)

# rnn.LSTM只传入输入embeddings，因此只返回最后一层的隐藏层在各时间步的隐藏状态。

# outputs形状是(词数, 批量大小, 2 \* 隐藏单元个数)

outputs, \_ = self.encoder(embeddings) # output, (h, c)

# 连结初始时间步和最终时间步的隐藏状态作为全连接层输入。它的形状为

# (批量大小, 4 \* 隐藏单元个数)。

encoding = torch.cat((outputs[0], outputs[-1]), -1)

outs = self.decoder(encoding)

return outs

4）编写目录调用的\_\_init\_\_.py文件

为了方便在掐目录调用BiRNN类，我们再model目录下创建\_\_init\_\_.py文件，并在该文件中编写代码如下：

**【代码12】\_\_init\_\_.py**

from .birnn import BiRNN as birnn

在BiLstm模型创建好之后，我们来编写实现训练过程的代码。在项目Sentiment-Analysis下新建Python File文件“main”。

1） 导入需要使用的包并加载配置参数

代码如下

【代码13】main.py

import argparse

from torch import optim

from torch import nn

import model

from configs import BasicConfigs

from process\_data import prepare\_vocab

import time

import torch

import numpy as np

bc = BasicConfigs()#加载配置参数

2） 使用argparse配置参数

因为要指定epochs和模型名称，我们需要使用argparse配置运行参数，参数有以下三个：

• compute\_val:是否输出验证集成绩；

• epochs：训练轮次；

• model\_name：模型名称

代码如下：

【代码14】main.py

# 获取参数

parser = argparse.ArgumentParser()

parser.add\_argument('--compute-val', default=True, action='store\_true', help='compute validation accuracy or not, default:None')

parser.add\_argument('--epoches', default=10, type=int, help='num of epoches for trainning loop, default:20')

parser.add\_argument('--model-name', default='birnn', help='choose one model name for trainng')

args = parser.parse\_args()

3） 编写实现训练流程的函数train

train函数实现训练集训练和验证集验证的流程，函数接收参数如下：

• net：模型网络，调用时使用实例化后的BiLstm网络进行赋值

• optimizer：优化器

• loss\_func：损失函数，后期传递的是交叉熵损失函数

• train\_iter：训练集数据迭代器

• val\_iter：验证集数据迭代器

• compute\_val：开关变量，控制是否输出验证集成绩

• device：设备，cpu或者gpu

• epochs：训练轮数

• load\_model\_dir：已训练模型的路径

• save\_model\_dir：模型保存路径

函数主要流程如下：

• 判断并加载已训练模型

• 遍历epochs

• 遍历数据集

• 计算输出值

• 计算损失

• 计算梯度

• 更新参数

• 每10步输出一次训练集成绩

• 每个epoch结束后，输出验证集成绩

首先，我们实现训练部分，编写代码如下：

【代码15】main.py

def train(net, optimizer, loss\_func, train\_iter, val\_iter, compute\_val,device, epoches, load\_model\_dir, save\_model\_dir):

print(f'>>>We are gonna tranning {net.\_\_class\_\_.\_\_name\_\_} with epoches of {epoches}<<<')

net = net.to(device)#将模型网络放入device

if load\_model\_dir:#如果给定加载模型路径，则加载模型

net.load\_state\_dict(torch.load(load\_model\_dir))

batch\_count = 0

for epoch in range(epoches):#遍历epochs

print(f'=>we are training epoch[{epoch+1}]...<=')

train\_l\_sum, train\_acc\_sum, n, start = 0.0, 0.0, 0, time.time()

for iter\_num, batch in enumerate(train\_iter):#遍历数据集数据

X,y = batch.text.to(device), batch.label.squeeze(0).to(device)#获取一批数据

score = net(X)#模型计算输出值

loss = loss\_func(score, y)#计算损失

optimizer.zero\_grad()#梯度清零

loss.backward()#反向传播求导

optimizer.step()#优化参数

train\_l\_sum += loss.cpu().item()#得到损失的值

train\_acc\_sum += (score.argmax(dim=1) == y).sum().cpu().item()#计算准确率

n += y.shape[0]

batch\_count += 1

train\_acc = train\_acc\_sum / n

if (iter\_num+1) % 10 == 0:#每10步输出结果

print("Train accuracy now is %.1f" % (round(train\_acc, 3)\*100)+'%') e

然后，在每个epoch结束后，我们需要对验证集数据进行预测并输出成绩，在所有的epoch结束后，保存模型。代码如下：

【代码16】main.py

if compute\_val:

net.eval()

val\_acc=[]

for iter\_num, batch in enumerate(val\_iter):#遍历验证集数据

val\_X = batch.text.to(device)

val\_y = batch.label.squeeze(0).to(device)

val\_score = net(val\_X)#模型输出

val\_acc.append((val\_score.argmax(dim=1) == val\_y).sum().cpu().item() / len(val\_y))#计算acc

print("Val accuracy is %.1f " % (round(np.mean(val\_acc), 3) \* 100) + '%')#输出成绩

net.train()

print('\*' \* 25)

if (epoch+1) % 5 ==0 and save\_model\_dir:#每5轮保存一次模型

print(f'saving model into => {save\_model\_dir}')

torch.save(net.state\_dict(), save\_model\_dir)

print('epoch %d, loss %.4f, train acc %.3f, time %.1f sec'

% (epoch + 1, train\_l\_sum / batch\_count, train\_acc, time.time() - start))

4） 编写代码实现整个训练过程。

实现了train函数后，我们需要准备好train函数需要的模型网络、损失函数、优化器并调用train函数实现模型训练和验证，代码如下：

【代码17】main.py

#构建词表、数据迭代器

TEXT,\_,train\_iter,val\_iter,vocab\_size=prepare\_vocab()

# 获取模型名称

net = getattr(model, args.model\_name)(TEXT=TEXT,vocab\_size=vocab\_size)

device = bc.device#获取设备

#构建优化器

optimizer = optim.Adam(net.parameters(), lr=bc.lr,weight\_decay=bc.alpha)

#构建损失函数

loss\_func = nn.CrossEntropyLoss()

load\_model\_dir=None

if \_\_name\_\_ == '\_\_main\_\_':

#调用train函数实现训练

train(net=net, optimizer=optimizer, loss\_func=loss\_func,

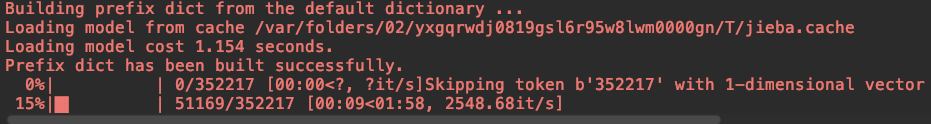
train\_iter=train\_iter, val\_iter=val\_iter,

compute\_val=args.compute\_val, device=device, epoches=args.epoches,

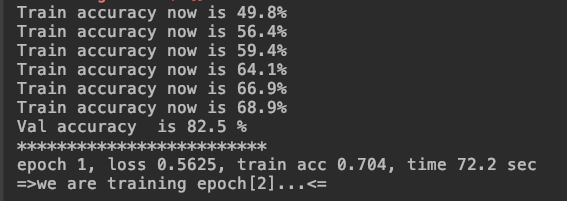
load\_model\_dir=load\_model\_dir, save\_model\_dir=bc.save\_model\_dir[args.model\_name])

运行main.py，进行模型训练，巡行过程如下所示：

正在进行文本分词和词表构建：



模型训练过程：



可以看到，经过1轮训练，训练集acc为0.704，验证集为0.825

模型训练完毕之后，我们需要使用测试集来对模型进行评估，下面我们就来编写进行模型评估的代码。在项目Sentiment-Analysis下新建Python File文件“evaluation.py”。

1） 导入需要使用的包并加载配置参数

【代码18】evaluation.py

import pandas as pd

import torch

import argparse

import model

from configs import BasicConfigs

from process\_data import transform\_data,prepare\_vocab

bc=BasicConfigs()#加载模型配置参数

2）使用argparse配置参数

因为要指定模型名称，我们需要使用argparse配置运行参数，参数有一个：

• model\_name：模型名称

编写代码如下：

【代码19】evaluation.py

parser = argparse.ArgumentParser()

parser.add\_argument('--model-name', default='birnn', help='choose one model name for trainng')

args = parser.parse\_args()

3）编写实现评估的函数evaluate

该函数功能为实现评估过程，接收一个df参数，即输入的测试集数据，计算过程如下：

• 加载词表并创建字段

• 实例化模型并加载模型参数

• 遍历测试数据每一行，预处理数据并使用模型进计算输出值

• 记录正确或者错误，计算并输出准确率

编写代码如下：

【代码20】evaluation.py

def evaluate(df):

# 加载字典

TEXT, LABEL, vocab\_size = prepare\_vocab(is\_train=False)

# 获取模型名称

net = getattr(model, args.model\_name)(TEXT=TEXT, vocab\_size=vocab\_size)

#加载模型参数

net.load\_state\_dict(torch.load(bc.save\_model\_dir[args.model\_name]))

result = {'correct':0, 'wrong':0}

df\_len = df.shape[0]

for i in range(df\_len):#遍历测试数据的每一行

record = df.loc[i, :].to\_dict()

data, label = transform\_data(record, TEXT, LABEL)

score = net(data)#模型计算输出值

if score.argmax(dim=1) == label:#判断是否正确

result['correct'] += 1

else:

result['wrong'] += 1

print(f"Classification Accuracy of Model({model.\_\_class\_\_.\_\_name\_\_})is {result['correct']/df\_len} ")

3）编写main代码实现评估

编写好evaluate函数后，我们需要编写主函数代码，读取数据并调用evaluate函数进行评估。

编写代码如下：

【代码21】evaluation.py

if \_\_name\_\_ == '\_\_main\_\_':

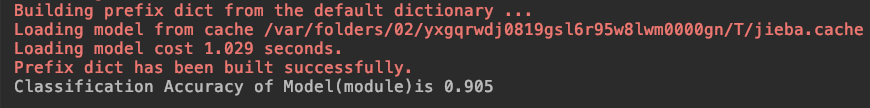
#读取数据

test\_data = pd.read\_csv('data/test.csv')

#评估

evaluate(df=test\_data)

代码编写完毕后，运行evaluation.py文件，运行结果如图所示：



7.4flask部署和调用

模型训练和评估完成之后，我们需要将模型部署上线，这里我们采用flask微服务框架来部署模型。我们将文本预处理、模型预测的整个过程封装到一个api接口中，提供给第三方人员调用。

封装接口为：/sentiment，接收一个get参数“sentence”，其值为需要进行情感分类的文本内容；

调用方式为：http://localhost:5000/sentiment?sentence=感觉还不错，液晶电视，配有电脑。位置很好

1）部署并封装接口。

该部分的流程如下：

• 实例化flask的app

• 加载词典、实例化模型并加载模型参数，这样词典和模型都导入到内存，随时可以接收数据并进行预测

• 编写api接口

• 启动app服务

首先，我们导入需要使用的包。加载模型配置参数

在项目Sentiment-Analysis下新建Python File文件“api.py”。

【代码22】api.py

import torch

import argparse

import model

from flask import Flask, request, jsonify

from process\_data import transform\_data,prepare\_vocab

from configs import BasicConfigs

bc = BasicConfigs()#加载模型配置参数

parser = argparse.ArgumentParser()

parser.add\_argument('--model-name', default='birnn', help='choose one model name for trainng')

args = parser.parse\_args()

然后，实例化flask的app，加载词典，实例化模型并加载模型参数：

【代码23】api.py

app = Flask(\_\_name\_\_)#实例化flask的app

app.config['JSON\_AS\_ASCII'] = False

#加载字典

TEXT, LABEL,vocab\_size=prepare\_vocab(is\_train=False)

# 获取模型名称

net = getattr(model, args.model\_name)(TEXT=TEXT,vocab\_size=vocab\_size)

#加载模型参数

net.load\_state\_dict(torch.load(bc.save\_model\_dir[args.model\_name]))

接着，编写实现接收请求的接口，在该接口中，我们主要实现以下功能：

• 提取请求中的文本数据

• 使用之前编写的函数transform\_data对文本进行分词、映射等预处理

• 模型进行预测

• 返回预测结果作为请求的响应

继续编写如下代码：

【代码24】api.py

@app.route('/sentiment')

def sentiemnt():

sentence = request.args.get('sentence')#提取请求中的文本数据

record = {'text':sentence}#封装成字典

data, \_ = transform\_data(record, TEXT, LABEL)#对文本进行预处理

idx = net(data).argmax(dim=1).item()#模型预测得到类别下标

prediction=LABEL.vocab.itos[idx]#下标转换成标签

if prediction=='pos':#判断标签

result = '积极'

else:

result = '消极'

return jsonify({'data':result, 'status\_code':200})#返回响应结果

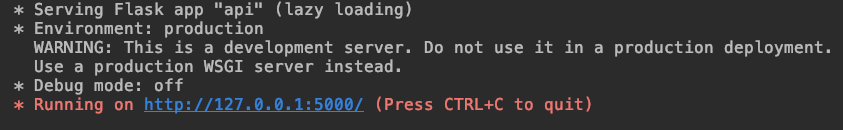
最后，编写main代码，启动app服务

【代码25】api.py

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=False)#启动app服务

运行api.py文件，结果如下图所示：



出现上图结果就表示运行成功，可以看到，服务已经启动，端口为5000。

2）编写测试接口的代码

为了测试接口是否能够正常使用，我们使用requests框架来发送get请求，测试接口能否正常返回预测结果。在项目Sentiment-Analysis下新建Python File文件“test\_api.py”。

编写代码如下：

【代码26】main.py

import requests

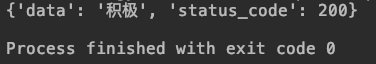
url='http://localhost:5000/sentiment'

sentence='感觉还不错，液晶电视，配有电脑。位置很好,'

res=requests.get(url,params={'sentence':sentence})#发送请求

print(res.json())

运行test\_api.py文件，结果如下图所示：



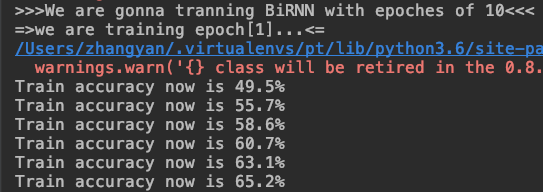
可以看到，对于“感觉还不错，液晶电视，配有电脑。位置很好”这个评论，模型预测为积极。

8.案例实验结果（结论）

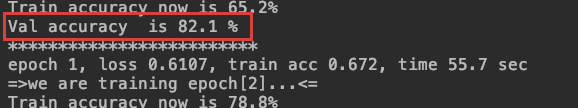
1）模型训练。

本次案例中，我们搭建了BiLstm模型，并对酒店评论文本进行情感二分类训练。经过10个轮次训练，最终在验证集上取得准确率0.894的结果。

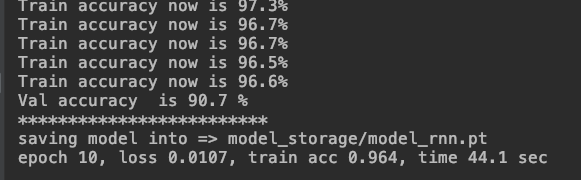
我们运行main.py文件，结果如下图所示：



每训练一个轮次，都会输出验证集评估结果，如下图所示：



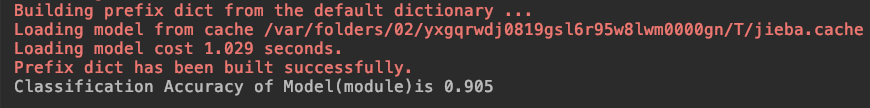
最终，经过10个轮次的训练，模型在验证集上达到了0.907的准确率，如下图所示：



2）模型评估。

模型训练结束后，我们使用测试集对模型进行评估。

我们运行evaluation.py文件，结果如下图所示：

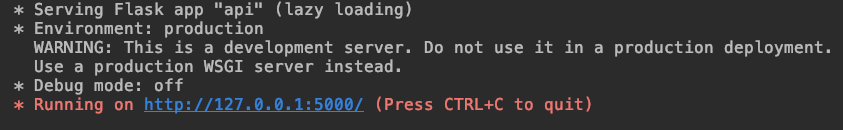


模型在测试集上达到了0.905的准确率。

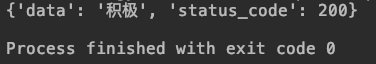
3）模型部署。

模型评估完成后，我们使用flask微服务框架来实现模型的部署，并开放ai接口提供给第三方调用。

我们首先运行api.py文件，启动服务，结果如下图所示：



在上图中国可以看出，服务已经启动，开放的端口为5000。下面我们测试接口，运行test\_api.py文件，得到如下的预测结果：



可以看到，对于“感觉还不错，液晶电视，配有电脑。位置很好”这个评论，模型预测为积极。

9.案例代码

【案例代码】configs.py

import torch

class BasicConfigs():

#数据存放参数

neg='data/neg'#负样本目录

pos='data/pos'#正样本目录

data\_path='data'#分割后数据存放目录

text\_vocab\_path='model\_storage/text.vocab'#文本词典存放目录

label\_vocab\_path ='model\_storage/label.vocab'#标签词典存放目录

stop\_word\_path='data/stopword.txt'#停用词文件路径

#词向量参数

embedding\_loc = 'data/sgns.wiki.word'#词向量文件路径

#模型训练参数

device = 'cuda' if torch.cuda.is\_available() else 'cpu'#设备

lr = 0.001#学习率

dropout\_rate = 0.5#随机失活比例

train\_embedding = True#是否训练词嵌入向量

batch\_size = 64#批次大小

alpha=0.001#L2惩罚项系数

#textcnn配置参数

kernel\_sizes = [3, 4, 5] # 3个 conv1d的size

num\_channels = [100, 100, 100]#卷积核个数

# bilstm配置参数

num\_hiddens = 100#lstm神经元数

num\_layers = 1#lstm层数

save\_model\_dir={#模型保存路径

'textcnn':'model\_storage/model\_cnn.pt',

'birnn':'model\_storage/model\_rnn.pt'

}

【案例代码】process\_data.py

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from configs import BasicConfigs

from glob import glob

import torch

import jieba

import pickle

from torchtext.vocab import Vectors

from torchtext.data import Field, TabularDataset, BucketIterator

import re

bc = BasicConfigs()

def load\_data\_to\_csv():

#遍历文件夹 读取数据

contents=[]

for file in glob(bc.neg+"/\*.txt"):#读取neg目录中的所有文件

with open(file,'r',encoding='utf-8') as f:

content=''.join([line.strip() for line in f.readlines()])#读取文本

contents.append([content,'neg'])#添加标签

for file in glob(bc.pos+"/\*.txt"):#读取pos目录中的所有文件

with open(file,'r',encoding='utf-8') as f:

content = ''.join([line.strip() for line in f.readlines()])

contents.append([content,'pos'])

#打乱顺序并存储到train.csv,test.csv,val.csv

#封装df

df=pd.DataFrame(contents,columns=['text','label'])

train,test=train\_test\_split(df,test\_size=0.1,random\_state=12)#数据分割

train,val=train\_test\_split(train,test\_size=0.2,random\_state=12)#训练集再分割

train.to\_csv(bc.data\_path+'/train.csv',index=False)#保存数据

val.to\_csv(bc.data\_path + '/val.csv', index=False)

test.to\_csv(bc.data\_path + '/test.csv', index=False)

print("文件处理完毕")

# 文本清洗

def clearTxt(line):

if line != '':

line = line.strip()

# 去除文本中的中文符号和英文符号

line = re.sub("[\s+\.\!\/\_,$%^\*(+\"\'；：“”．]+|[+——！，。？?、~@#￥%……&\*（）]+", "", line)

return line

def my\_cut(line):

line=clearTxt(line)#清洗

return jieba.lcut(line)#分词并返回

def transform\_data(record, TEXT, LABEL):

if not isinstance(record, dict):

raise ValueError('Make sure data is dict')

tokens = my\_cut(record['text'])#文本清洗和分词

res = []

for token in tokens:#遍历每一个词，进行映射

res.append(TEXT.vocab.stoi[token])

data = torch.tensor(res).unsqueeze(1)

if 'label' in list(record):#如果有标签

#对标签进行映射

label = torch.tensor(LABEL.vocab.stoi[record['label']])

else:

label = None

return data, label

def prepare\_vocab(is\_train=True):

# 定义Field

TEXT = Field(tokenize=my\_cut)

LABEL = Field(eos\_token=None, pad\_token=None, unk\_token=None)

# 定义字段与FIELD之间读配对

fields = [('text', TEXT), ('label', LABEL)]

if is\_train:

# 读取数据

train\_data, val\_data = TabularDataset.splits(path=bc.data\_path, train='train.csv',

validation='val.csv',

format='csv',

fields=fields,

skip\_header=True)

# 构建从本地加载的词向量

vectors = Vectors(name=bc.embedding\_loc)

# 词表和标签映射表

TEXT.build\_vocab(train\_data, val\_data, vectors=vectors)

LABEL.build\_vocab(train\_data, val\_data)

#创建训练集和验证集的批次迭代器

train\_iter = BucketIterator(train\_data,

batch\_size=bc.batch\_size,

sort\_key=lambda x: len(x.text),

sort\_within\_batch=True,

shuffle=True)

val\_iter = BucketIterator(val\_data,

batch\_size=bc.batch\_size,

sort\_key=lambda x: len(x.text),

sort\_within\_batch=True,

shuffle=True)

#保存词表和标签映射表

with open(bc.text\_vocab\_path, 'wb')as f:

pickle.dump(TEXT.vocab, f)

with open(bc.label\_vocab\_path, 'wb')as f:

pickle.dump(LABEL.vocab, f)

vocab\_size = TEXT.vocab.vectors.shape

return TEXT,LABEL,train\_iter,val\_iter,vocab\_size

else:

# 加载词典

with open(bc.text\_vocab\_path, 'rb')as f:

TEXT.vocab = pickle.load(f)#词典配置到字段

#加载标签词表

with open(bc.label\_vocab\_path, 'rb')as f:

LABEL.vocab = pickle.load(f)#配置

vocab\_size = TEXT.vocab.vectors.shape#得到词表形状

return TEXT,LABEL,vocab\_size

if \_\_name\_\_ == '\_\_main\_\_':

load\_data\_to\_csv()

【案例代码】model/birnn.py

import torch

from torch import nn

from configs import BasicConfigs

bc = BasicConfigs()

class BiRNN(nn.Module):

def \_\_init\_\_(self,TEXT,vocab\_size, num\_hiddens=bc.num\_hiddens, num\_layers=bc.num\_layers):

super(BiRNN, self).\_\_init\_\_()

#嵌入层

self.embedding = nn.Embedding(vocab\_size[0], vocab\_size[1])

#加载词向量权重

self.embedding.weight.data.copy\_(TEXT.vocab.vectors)

# bidirectional设为True即得到双向循环神经网络

self.encoder = nn.LSTM(input\_size=vocab\_size[1],

hidden\_size=num\_hiddens,

num\_layers=num\_layers,

bidirectional=True)

# 初始时间步和最终时间步的隐藏状态作为全连接层输入

self.decoder = nn.Linear(4\*num\_hiddens, 2)

def forward(self, inputs):

# 再提取词特征，输出形状为(词数, 批量大小, 词向量维度)

embeddings = self.embedding(inputs)

# rnn.LSTM只传入输入embeddings，因此只返回最后一层的隐藏层在各时间步的隐藏状态。

# outputs形状是(词数, 批量大小, 2 \* 隐藏单元个数)

outputs, \_ = self.encoder(embeddings) # output, (h, c)

# 连结初始时间步和最终时间步的隐藏状态作为全连接层输入。它的形状为

# (批量大小, 4 \* 隐藏单元个数)。

encoding = torch.cat((outputs[0], outputs[-1]), -1)

outs = self.decoder(encoding)

return outs

【案例代码】model/\_\_init\_\_.py

from .birnn import BiRNN as birnn

【案例代码】main.py

import argparse

from torch import optim

from torch import nn

import model

from configs import BasicConfigs

from process\_data import prepare\_vocab

import time

import torch

import numpy as np

bc = BasicConfigs()

# 获取参数

parser = argparse.ArgumentParser()

parser.add\_argument('--compute-val', default=True, action='store\_true', help='compute validation accuracy or not, default:None')

parser.add\_argument('--epoches', default=10, type=int, help='num of epoches for trainning loop, default:20')

parser.add\_argument('--model-name', default='birnn', help='choose one model name for trainng')

args = parser.parse\_args()

def train(net, optimizer, loss\_func, train\_iter, val\_iter, compute\_val,device, epoches, load\_model\_dir, save\_model\_dir):

print(f'>>>We are gonna tranning {net.\_\_class\_\_.\_\_name\_\_} with epoches of {epoches}<<<')

net = net.to(device)#将模型网络放入device

if load\_model\_dir:#如果给定加载模型路径，则加载模型

net.load\_state\_dict(torch.load(load\_model\_dir))

batch\_count = 0

for epoch in range(epoches):#遍历epochs

print(f'=>we are training epoch[{epoch+1}]...<=')

train\_l\_sum, train\_acc\_sum, n, start = 0.0, 0.0, 0, time.time()

for iter\_num, batch in enumerate(train\_iter):#遍历数据集数据

X,y = batch.text.to(device), batch.label.squeeze(0).to(device)#获取一批数据

score = net(X)#模型计算输出值

loss = loss\_func(score, y)#计算损失

optimizer.zero\_grad()#梯度清零

loss.backward()#反向传播求导

optimizer.step()#优化参数

train\_l\_sum += loss.cpu().item()#得到损失的值

train\_acc\_sum += (score.argmax(dim=1) == y).sum().cpu().item()#计算准确率

n += y.shape[0]

batch\_count += 1

train\_acc = train\_acc\_sum / n

if (iter\_num+1) % 10 == 0:#每10步输出结果

print("Train accuracy now is %.1f" % (round(train\_acc, 3)\*100)+'%')

if compute\_val:

net.eval()

val\_acc=[]

for iter\_num, batch in enumerate(val\_iter):#遍历验证集数据

val\_X = batch.text.to(device)

val\_y = batch.label.squeeze(0).to(device)

val\_score = net(val\_X)#模型输出

val\_acc.append((val\_score.argmax(dim=1) == val\_y).sum().cpu().item() / len(val\_y))#计算acc

print("Val accuracy is %.1f " % (round(np.mean(val\_acc), 3) \* 100) + '%')#输出成绩

net.train()

print('\*' \* 25)

if (epoch+1) % 5 ==0 and save\_model\_dir:#每5轮保存一次模型

print(f'saving model into => {save\_model\_dir}')

torch.save(net.state\_dict(), save\_model\_dir)

print('epoch %d, loss %.4f, train acc %.3f, time %.1f sec'

% (epoch + 1, train\_l\_sum / batch\_count, train\_acc, time.time() - start))

#构建词表、数据迭代器

TEXT,\_,train\_iter,val\_iter,vocab\_size=prepare\_vocab()

# 获取模型名称

net = getattr(model, args.model\_name)(TEXT=TEXT,vocab\_size=vocab\_size)

device = bc.device#获取设备

#构建优化器

optimizer = optim.Adam(net.parameters(), lr=bc.lr,weight\_decay=bc.alpha)

#构建损失函数

loss\_func = nn.CrossEntropyLoss()

load\_model\_dir=None

if \_\_name\_\_ == '\_\_main\_\_':

#调用train函数实现训练

train(net=net, optimizer=optimizer, loss\_func=loss\_func,

train\_iter=train\_iter, val\_iter=val\_iter,

compute\_val=args.compute\_val, device=device, epoches=args.epoches,

load\_model\_dir=load\_model\_dir, save\_model\_dir=bc.save\_model\_dir[args.model\_name])

【案例代码】evaluation.py

'''

Evaluate model performence

'''

import pandas as pd

import torch

import argparse

import model

from configs import BasicConfigs

from process\_data import transform\_data,prepare\_vocab

bc=BasicConfigs()

parser = argparse.ArgumentParser()

parser.add\_argument('--model-name', default='birnn',choices=['textcnn', 'birnn'], help='choose one model name for trainng')

args = parser.parse\_args()

def evaluate(df):

# 加载字典

TEXT, LABEL, vocab\_size = prepare\_vocab(is\_train=False)

# 获取模型名称

net = getattr(model, args.model\_name)(TEXT=TEXT, vocab\_size=vocab\_size)

#加载模型参数

net.load\_state\_dict(torch.load(bc.save\_model\_dir[args.model\_name]))

result = {'correct':0, 'wrong':0}

df\_len = df.shape[0]

for i in range(df\_len):#遍历测试数据的每一行

record = df.loc[i, :].to\_dict()

data, label = transform\_data(record, TEXT, LABEL)

score = net(data)#模型计算输出值

if score.argmax(dim=1) == label:#判断是否正确

result['correct'] += 1

else:

result['wrong'] += 1

print(f"Classification Accuracy of Model({model.\_\_class\_\_.\_\_name\_\_})is {result['correct']/df\_len} ")

if \_\_name\_\_ == '\_\_main\_\_':

#读取数据

test\_data = pd.read\_csv('data/test.csv')

#评估

evaluate(df=test\_data)

【案例代码】api.py

import torch

import argparse

import model

from flask import Flask, request, jsonify

from process\_data import transform\_data,prepare\_vocab

from configs import BasicConfigs

bc = BasicConfigs()#加载模型配置参数

parser = argparse.ArgumentParser()

parser.add\_argument('--model-name', default='birnn', help='choose one model name for trainng')

args = parser.parse\_args()

app = Flask(\_\_name\_\_)#实例化flask的app

app.config['JSON\_AS\_ASCII'] = False

#加载字典

TEXT, LABEL,vocab\_size=prepare\_vocab(is\_train=False)

# 获取模型名称

net = getattr(model, args.model\_name)(TEXT=TEXT,vocab\_size=vocab\_size)

#加载模型参数

net.load\_state\_dict(torch.load(bc.save\_model\_dir[args.model\_name]))

@app.route('/sentiment')

def sentiemnt():

sentence = request.args.get('sentence')#提取请求中的文本数据

record = {'text':sentence}#封装成字典

data, \_ = transform\_data(record, TEXT, LABEL)#对文本进行预处理

idx = net(data).argmax(dim=1).item()#模型预测得到类别下标

prediction=LABEL.vocab.itos[idx]#下标转换成标签

if prediction=='pos':#判断标签

result = '积极'

else:

result = '消极'

return jsonify({'data':result, 'status\_code':200})#返回响应结果

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=False)#启动app服务

【案例代码】test\_api.py

import requests

url='http://localhost:5000/sentiment'

sentence='感觉还不错，液晶电视，配有电脑。位置很好,'

res=requests.get(url,params={'sentence':sentence})#发送请求

print(res.json())