Text Classification

爬取豆瓣“最热”、“最新”、“评价最高”三个主题的电影评价进行文本分类

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  | |
|  |  |  |
|  |  | |
| **2019年7月21日** | | |

1. 实验环境
   1. 实验工具

Python 3.6.8、TensorFlow 1.13.1、beautifulsoup4 4.7.1、pandas 0.24.2、Numpy 1.14.1、jieba 0.39、torch 1.1.0 、nltk 3.4.3、metrics 0.3.3等

* 1. 实验平台
* Windows 10、NVIDIA GeForce GTX 1050 2G、CPU i7-8700、Mem 16 G
* Google Colab

1. 实验数据集
   1. 数据爬取

本实验数据集为来自我们自己从豆瓣爬取的电影短评。我们分别对“热门”、“最新”、“评价最高”三个主题的电影短评进去爬取数据，这三个主题的电影列表可从下列网址获取：

|  |  |
| --- | --- |
| 热门 | https://movie.douban.com/j/search\_subjects?type=movie&tag=%E7%83%AD%E9%97%A8&sort=recommend&page\_limit=500&page\_start=0 |
| 最新 | https://movie.douban.com/j/search\_subjects?type=movie&tag=%E6%9C%80%E6%96%B0&sort=recommend&page\_limit=500&page\_start=0 |
| 评价最高 | https://movie.douban.com/j/search\_subjects?type=movie&tag=%E8%B1%86%E7%93%A3%E9%AB%98%E5%88%86&sort=recommend&page\_limit=500&page\_start=0 |

我们分别爬取这三个主题的前500部电影，其中“热门”主题电影只用324部。由于豆瓣限制，对登录的用户，每部电影最多可查看25页\*20=500条短评。对未登录的用户，每部电影最多可查看10页\*20=500条短评。

针对每部电影的评价，可由以下方法生成网址：

# 用于生成短评页面网址的函数

def MakeUrl(mian\_url,start):

"""make the next page's url"""

url = mian\_url + '/comments?start=' + str(start) + '&limit=20&sort=new\_score&status=P'

print(url)

return url

由于豆瓣账号和电脑网络IP被封次数太多，放弃了登录爬取短评，使用未登录爬取。故最终获得了大约（500+500+324）\*10\*20=264800条短评。此外，部分短评没有评分，并且有重复的短评。故去除无评分和重复数据外，还剩下246045条数据来作为语料库。

对每条评论，我们获取了用户名称、评分、评价内容、发布时间、点赞数。对评分值，我们将50,40,30,20,10，分别设置为5,4,3,2,1，即{5:"力荐",4:"推荐",3:"还行",2:"较差",1:"很差"}。代码内容如下：

# 获得评论

soup = BeautifulSoup(res\_comment, "html.parser")

comments = soup.select("div.comment-item")

# 判断页面为空

if len(comments) == 0 or len(comments[0].select("div.avatar > a")) == 0:

print("Comments empty!")

checkpoint['id\_page']=page\_id

checkpoint['id\_item']=id\_item

return -1

for item\_id in range(id\_item, len(comments)):

item = comments[item\_id]

username = ''

score = ''

comment = ''

pub\_time = ''

votes = ''

# 获取评价人姓名

item\_username = item.select("div.avatar > a")

if len(item\_username) > 0:

username = item\_username[0]['title']

# 获取点赞数

item\_vote = item.select('div.comment > h3 > span.comment-vote > span[class="votes"]')

if len(item\_vote) > 0:

votes = item\_vote[0].get\_text()

# 获取评价值

# {5:"力荐",4:"推荐",3:"还行",2:"较差",1:"很差"}

temp\_score = item.select("div.comment > h3 > span.comment-info > span")

str\_score = ''

for ts in temp\_score:

str\_score += str(ts)

pattern = re.compile(r'<span class="allstar(.\*?) rating" title="(.\*?)"></span>')

patter\_score = pattern.findall(str\_score)

if patter\_score != []:

score = str(int(patter\_score[0][0])//10)

# 获取评价内容

temp\_comment = item.select('div.comment > p > span[class="short"]')

if len(temp\_comment) > 0:

comment = temp\_comment[0].get\_text()

# 去掉多余空白

comment = re.sub(r'\s+',' ', comment)

"""

# 只保留中文

comment = re.sub('[^\u4e00-\u9fa5]', '', comment)

# 分词，去掉停用词

temp\_words = jieba.lcut(comment)

comment = ' '.join(list(set(temp\_words)-set(stopwords)))

"""

# 获取评价时间

temp\_time = item.select('div.comment > h3 > span.comment-info > span[class="comment-time"]')

if len(temp\_time) > 0:

pub\_time = temp\_time[0]['title']

* 1. 数据清洗
* 首先将三个主题的数据何必，提取评分和评价内容这两列，其中评分均减一
* 评价内容去重，并去除评分为空的行，数据分布如下：

RangeIndex: 197900 entries, 0 to 197899

Data columns (total 2 columns):

score 197900 non-null object

content 197900 non-null object

memory usage: 3.0+ MB

0 0.040051

1 0.112936

4 0.238605

2 0.281885

3 0.326523

* 对每条评论只保留中文和英文，其中中文繁体字转为简体字，英文字母全部小写。同时对每条评论使用jieba分词，去掉停用词，去掉词数小于5的行，数据分布如下：

Int64Index: 147713 entries, 0 to 197898

Data columns (total 2 columns):

score 147713 non-null object

content 147713 non-null object

memory usage: 3.4+ MB

0 0.035251

1 0.105915

4 0.244535

2 0.275047

3 0.339252

* 针对二分类情况，为了使正向数据和反向数据比例均衡，将score>2的归为一类:正向评价（0、1、2）；score<=2的归为一类：负面评价（3、4）
* 语料库按9:1的比例划分为训练集和测试集，在将训练集按9:1的比例划分为训练集和验证集，其中训练集、验证集、测试集的0和1的比例如下：

train: 0.5827440950804874

val: 0.5898458066942459

test: 0.586785810993772

1. 实验方法
2. 基于词向量的TextCNN
   1. 构建词汇表

对jieba分词后的到的词语，去掉通用词，按词频排序，取前vocab\_size个，且加入“<pad>“、”<unk>“、”<eos>“。代码如下：

# 构建词汇表

def build\_word\_dict(train\_df, val\_df, out\_path, stopwords\_path, vocab\_size=0):

# 读取停用词

stopwords = [line.strip() for line in open(stopwords\_path, encoding='utf-8').readlines()]

df = [train\_df["content"], val\_df["content"]]

max\_sen\_len = 0

sum\_sen\_len = 0

words = list()

for contents in df:

for content in contents:

if content.strip():

token = jieba.lcut(content.strip())

token = [i for i in token if i not in stopwords and i.strip()]

# print(token)

sum\_sen\_len += len(token)

max\_sen\_len = max(max\_sen\_len, len(token))

for word in token:

words.append(word)

print('>>> ave\_sen\_len:'+str(sum\_sen\_len/(len(df[0])+len(df[0]))))

print('>>> max\_sen\_len:'+str(max\_sen\_len))

# 统计词频

word\_counter = collections.Counter(words)

if vocab\_size!=0 and len(word\_counter) > vocab\_size-3:

word\_counter=word\_counter.most\_common(vocab\_size-3)

else:

word\_counter = word\_counter.most\_common()

word\_dict = dict()

word\_dict["<pad>"] = 0

word\_dict["<unk>"] = 1

word\_dict["<eos>"] = 2

for word, \_ in word\_counter:

word\_dict[word] = len(word\_dict)

vocab\_size = len(word\_dict)

print('>>> vocab\_size:'+str(vocab\_size))

with open(out\_path, 'w', encoding='utf-8') as wd:

for w in word\_dict:

wd.write(w+'\t')

wd.write(str(word\_dict[w]))

wd.write('\n')

return word\_dict,max\_sen\_len,vocab\_size

* 1. 加载数据集，使用词向量表示

给每条句子最后加上“<eos>“，句子长度为max\_sen\_len。若不够，在句尾添加”<pad>“，对于不知道的词语，记作”<unk>“。对于socre使用one-hot表示。代码如下：

# 加载语料库：train/val/test

# x为构成一条评论的词所对应的id。 y为one-hot表示pos-[0, 1], neg-[1, 0]

# {4:"力荐"-[0,0,0,0,1],3:"推荐"-[0,0,0,1,0],2:"还行"-[0,0,1,0,0],1:"较差"-[0,1,0,0,0],0:"很差"-[0,0,0,0,0]}

def build\_word\_dataset(df\_data, word\_dict, stopwords\_path, max\_sen\_len=870, num\_class=2):

# Shuffle dataframe

df = df\_data.sample(frac=1)

# 读取停用词

stopwords = [line.strip() for line in open(stopwords\_path, encoding='utf-8').readlines()]

x = [[j for j in jieba.lcut(df.iat[i,1].strip()) if j not in stopwords and j.strip()] for i in df.index]

x = list(map(lambda d: list(map(lambda w: word\_dict.get(w, word\_dict["<unk>"]), d)), x))

x = list(map(lambda d: d + [word\_dict["<eos>"]], x))

x = list(map(lambda d: d[:max\_sen\_len], x))

x = list(map(lambda d: d + (max\_sen\_len - len(d)) \* [word\_dict["<pad>"]], x))

y = list(map(lambda d: [1 if i==d else 0 for i in range(num\_class)], list(df["score"])))

return x, y

* 1. 模型构建

Text-CNN和传统的CNN结构类似，具有词嵌入层、卷积层、池化层和全连接层的四层结构。

其中，Text-CNN的词嵌入层使用二维矩阵来表示长文本。词嵌入（Word embedding）将输入文本的每个词语通过空间映射，将独热表示（One-Hot Representation）转换成分布式表示（Distributed Representation），进而可以使用低维的词向量来表示每一个词语。经过词嵌入，每个单词具有相同长度的词向量表示。将各个词语的向量表示连起来便可以得到二维矩阵。得到词向量的方式有多种，常用的是Word2vec方法。此外，在训练模型的时候可以设置update\_w2v来选择更新或不更新词向量。

Text-CNN的卷积层是主要部分，卷积核的宽度等于词向量的维度，经卷积后可以提取文本的特征向量。和在图像领域应用类似，Text-CNN可以设置多个卷积核以提取文本的多层特征，长度为N的卷积核可以提取文本中的N-gram特征。

Text-CNN的池化层一般采取Max-over-time pooling，输出最大值，从而判断词嵌入中是否含N-gram。Text-CNN的全连接层采用了Dropout算法以防过拟合，并使用Softmax函数输出各个类别的概率。

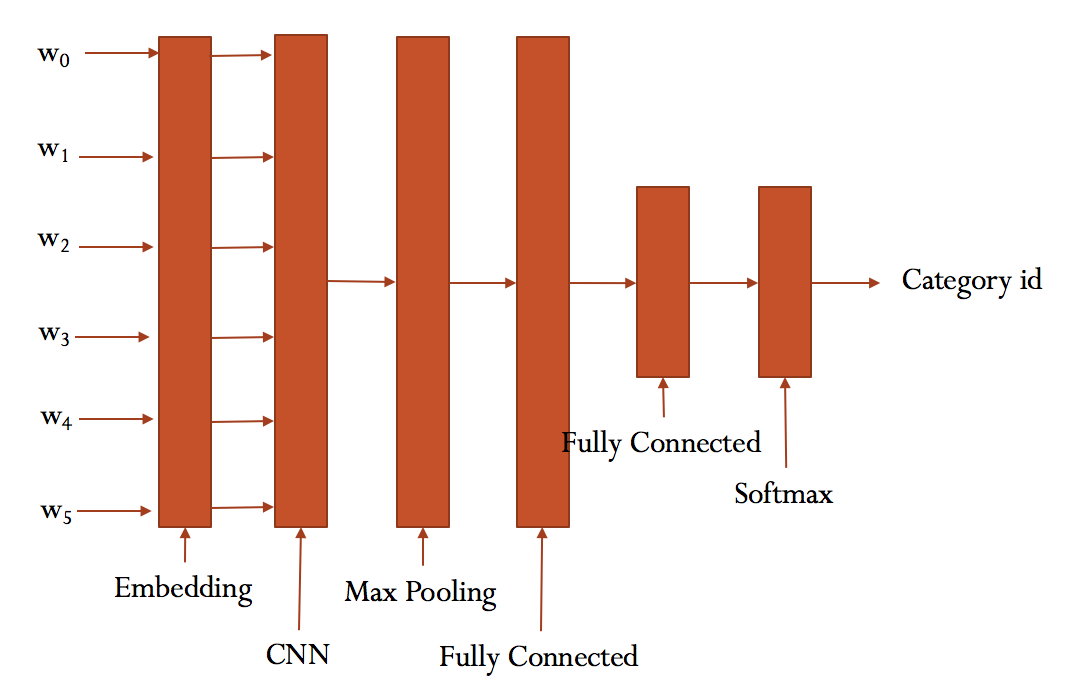


图1 TextCNN架构图

1. 基于TFIDF的神经网络
   1. 加载数据集，计算TF-IDF

在计算TF-IDF时，设置了max\_df和min\_df，删除超过max\_df比例的文档中出现的关键词（过于平凡）和低于这一数量的文档中出现的关键词（过于独特）代码如下：

def build\_tfidvec\_dataset(df\_data, stopwords\_path, n\_dim=20000, num\_class=2):

# 读取停用词

stopwords = [line.strip() for line in open(stopwords\_path, encoding='utf-8').readlines()]

# 通过jieba分词

def word\_clean(content):

return ' '.join([j for j in jieba.lcut(content.strip()) if j not in stopwords and j.strip()])

df\_data['content\_cut'] = df\_data.content.apply(word\_clean)

score = df\_data.iloc[:, 0].values

content = df\_data.iloc[:, 2].values

# 数据的TF-IDF信息计算

# sublinear\_tf=True 时生成一个近似高斯分布的特征，可以提高大概1~2个百分点

vectorizer = TfidfVectorizer(max\_df=0.95, min\_df=2, smooth\_idf=True, sublinear\_tf=True)

# 对数据训练

vec\_data = vectorizer.fit\_transform(content)

print(len(vectorizer.get\_feature\_names()))

return vec\_data, score

* 1. 模型构建

本模型主要采用nn.Linear线性变换、nn.ReLu激活函数、nn.Dropout堆叠，共4层。同时采用Adam来优化参数。

class TextNN(nn.Module):

def \_\_init\_\_(self, config):

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

self.config = config

self.classifier = nn.Sequential(

nn.Linear(config.n\_dim, 1024),

nn.ReLU(inplace=True),

nn.Dropout(),

nn.Linear(1024, 1024),

nn.ReLU(inplace=True),

nn.Dropout(),

nn.Linear(1024, 1024),

nn.ReLU(inplace=True),

nn.Dropout(),

nn.Linear(1024, 512),

nn.ReLU(inplace=True),

nn.Dropout(),

nn.Linear(512, config.num\_classes)

)

self.optimizer = torch.optim.Adam(self.parameters(), lr=config.learning\_rate)

self.exp\_lr\_scheduler = lr\_scheduler.StepLR(self.optimizer, step\_size=int(config.num\_epochs/2), gamma=config.gamma)

self.losser = nn.CrossEntropyLoss()

def forward(self, x):

output = self.classifier(x.double())

return output.squeeze(1)

1. 朴素贝叶斯
   1. 加载数据集，计算TF-IDF

针对TF-IDF的计算与基于TF-IDF的神经网络类似，此外这里采用了管道的方式来读取和处理数量（英文内存不够了），此外数据集划分为训练集和测试集：4:1.

#划分训练集和测试集

x\_train,x\_test,y\_train,y\_test = train\_test\_split(x, y, test\_size=0.2, random\_state=20)

nb = MultinomialNB()

# 读取停用词

stopwords = [line.strip() for line in open('data/stopwords.txt', encoding='utf-8').readlines()]

max\_df=0.8 # 在超过这一比例的文档中出现的关键词（过于平凡），去除掉

min\_df=3 # 在低于这一数量的文档中出现的关键词（过于独特），去除掉

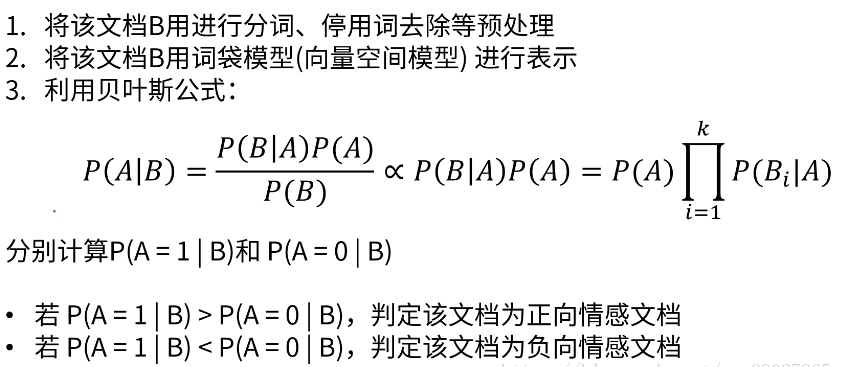
vect=CountVectorizer(max\_df=max\_df,min\_df=min\_df,stop\_words=frozenset(stopwords))

# 利用管道顺序连接工作

pipe=make\_pipeline(vect,nb)

* 1. 模型构建

朴素贝叶斯模型：



1. 实验步骤与结果
2. 在工程目录下，运行指令：

models = ['textcnn', 'textrnn', 'textcnn\_run','textcnn\_2l','bayes', 'text\_nn']

sets = ['train', 'test', 'predict']

python run.py –model=text\_nn –set=train

其中’predict’操作为单个句子预测，效果如下：



图2 text\_nn预测结果

1. 参数设置

class TextConfig(object):

"""Text配置参数"""

update\_w2v = True # 是否在训练中更新w2v

embedding\_dim = 50 # 词向量维度

max\_sen\_len = 870 # 序列长度 1145

vocab\_size = 145346 # 词汇大小 56366

num\_classes = 2 # 类别数

num\_filters = 256 # 卷积核数目

filter\_sizes = [3,4,5] # 卷积核尺寸

kernel\_size=4

window\_size=20 #感受野大小

pooling\_window=4

pooling\_stride=2

num\_layers= 2 # 隐藏层层数

hidden\_dim = 256 # 隐藏层神经元

rnn = 'lstm' # lstm 或 gru

dropout\_keep\_prob = 0.5 # dropout保留比例

learning\_rate = 1e-3 # 学习率

batch\_size = 64 # 每批训练大小

num\_epochs = 100 # 总迭代轮次,即遍历整个训练样本的次数

print\_per\_batch = 10 # 每多少轮输出一次结果

save\_per\_batch = 10 # 每多少轮存入tensorboard

require\_improvement = 1000 # 如果超过1000轮未提升，提前结束训练

max\_df=0.95 # 在超过这一比例的文档中出现的关键词（过于平凡），去除掉

min\_df=2 # 在低于这一数量的文档中出现的关键词（过于独特），去除掉

n\_dim = 68268 #数据维度 0.8 3 50934

gamma = 0.1

save\_dir = './checkpoints/' # 训练模型保存的地址

data\_path = './data/comment.csv'

stopwords\_path = './data/stopwords.txt'

train\_path = './data/train.txt'

dev\_path = './data/val.txt'

test\_path = './data/test.txt'

word\_dict\_path = './data/word\_dict.txt'

pre\_word2vec\_path = './data/wiki\_word2vec\_50.bin'

word2vec\_path = './data/word\_vecs.txt'

1. 模型训练

* TextCNN

训练总共迭代整个数据集num\_epochs次，每批训练大小batch\_size。每print\_per\_batch轮，在训练集和训练集上验证性能，并保存最好性能的模型。若超过require\_improvement轮性能为提升，则提前结束训练，代码如下：

# 训练

def train(word\_dict, model):

print("Configuring TensorBoard and Saver...")

# 配置 Tensorboard，重新训练时，请将tensorboard文件夹删除，不然图会覆盖

tensorboard\_dir = 'tensorboard/'+args.model

if not os.path.exists(tensorboard\_dir):

os.makedirs(tensorboard\_dir)

tf.summary.scalar("loss", model.loss)

tf.summary.scalar("accuracy", model.accuracy)

merged\_summary = tf.summary.merge\_all()

writer = tf.summary.FileWriter(tensorboard\_dir)

# 配置 Saver

saver = tf.train.Saver()

if not os.path.exists(config.save\_dir):

os.makedirs(config.save\_dir)

# 载入训练集与验证集

start\_time = time.time()

print('加载train语料库========================')

train\_df = pd.read\_csv(config.train\_path, sep='\t', header=None, names=["score", "content"], encoding='utf-8')

x\_train, y\_train = build\_word\_dataset(

train\_df, word\_dict, config.stopwords\_path, max\_sen\_len=config.max\_sen\_len, num\_class=config.num\_classes)

print('加载Validation语料库===================')

val\_df = pd.read\_csv(config.dev\_path, sep='\t', header=None, names=["score", "content"], encoding='utf-8')

x\_val, y\_val = build\_word\_dataset(

val\_df, word\_dict, config.stopwords\_path, max\_sen\_len=config.max\_sen\_len, num\_class=config.num\_classes)

time\_dif = get\_time\_dif(start\_time)

print("Time usage:", time\_dif)

# 创建session

sess = tf.Session()

sess.run(tf.global\_variables\_initializer())

writer.add\_graph(sess.graph)

print('Training and evaluating===============')

start\_time = time.time()

total\_batch = 0 # 总批次

best\_acc\_val = 0.0 # 最佳验证集准确率

last\_improved = 0 # 记录上一次提升批次

flag = False

for epoch in range(config.num\_epochs):

print('Epoch:', epoch + 1)

for train\_feed, train\_n in model.get\_batches(x\_train, y\_train, batch\_size=config.batch\_size):

if total\_batch % config.save\_per\_batch == 0:

# 每多少轮次将训练结果写入tensorboard scalar

train\_feed[model.is\_training]=True

s = sess.run(merged\_summary, feed\_dict=train\_feed)

writer.add\_summary(s, total\_batch)

if total\_batch % config.print\_per\_batch == 0:

# 每config.print\_per\_batch轮次输出在训练集和验证集上的性能

loss\_train, acc\_train = model.train\_on\_batch(sess, train\_feed)

loss\_val, acc\_val = model.evaluate(sess, x\_val, y\_val)

if acc\_val >= best\_acc\_val:

# 保存在验证集上性能最好的模型

best\_acc\_val = acc\_val

last\_improved = total\_batch

saver.save(sess=sess, save\_path=os.path.join(config.save\_dir, 'best\_text\_cnn\_model'))

improved\_str = '\*'

else:

improved\_str = ''

time\_dif = get\_time\_dif(start\_time)

msg = 'Iter: {0:>6}, Train Loss: {1:>6.4}, Train Acc: {2:>7.4%},' + \

' Val Loss: {3:>6.4}, Val Acc: {4:>7.4%}, Time: {5} {6}'

print(msg.format(total\_batch, loss\_train, acc\_train, loss\_val, acc\_val, time\_dif, improved\_str))

# 运行优化

train\_feed[model.is\_training]=True

sess.run(model.optimizer, feed\_dict=train\_feed)

total\_batch += 1

if total\_batch - last\_improved > config.require\_improvement:

# 验证集正确率长期不提升，提前结束训练

print("No optimization for a long time, auto-stopping...")

flag = True

break # 跳出循环

if flag: # 同上

break

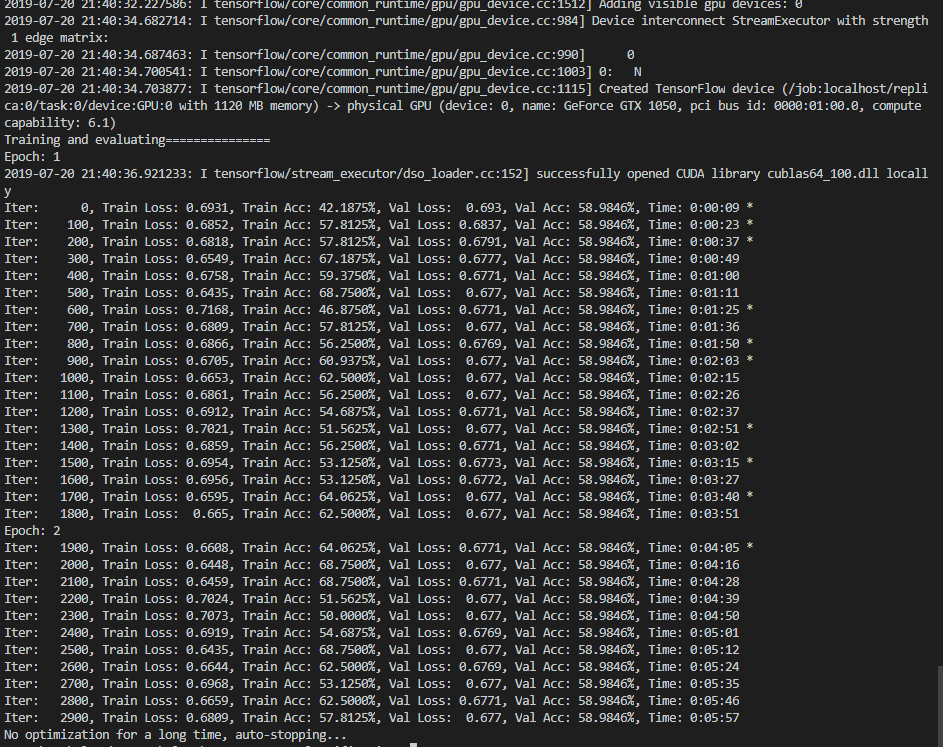


图3 TextCNN-5训练结果

* 基于TF-IDF的神经网络

训练总共迭代整个数据集num\_epochs次，每批训练大小batch\_size。每迭代一次，在验证集上验证性能，并保存最好性能的模型。

def nn\_train():

# 载入训练集与验证集

start\_time = time.time()

print('加载train语料库========================')

train\_df = pd.read\_csv(config.train\_path, sep='\t', header=None, names=["score", "content"], encoding='utf-8')

print('加载Validation语料库===================')

val\_df = pd.read\_csv(config.dev\_path, sep='\t', header=None, names=["score", "content"], encoding='utf-8')

x\_train, y\_train, x\_val, y\_val, vectorizer = build\_tfidvec\_dataset(train\_df, val\_df, config.stopwords\_path, max\_df=config.max\_df, min\_df=config.min\_df, num\_class=config.num\_classes)

config.n\_dim = len(vectorizer.get\_feature\_names())

model = TextNN(config)

device = torch.device("cuda:0" if torch.cuda.is\_available() else "cpu")

model.to(device)

train\_dataloader, \_train\_len = model.dataloader(x\_train,y\_train,config.batch\_size,shuffle=True,num\_workers=1)

val\_dataloader, \_val\_len = model.dataloader(x\_val,y\_val,config.batch\_size,shuffle=False,num\_workers=1)

model = model.double()

# 保存准确度最高的模型

best\_model = copy.deepcopy(model)

best\_accuracy = 0.0

for epoch in range(config.num\_epochs):

model.exp\_lr\_scheduler.step()

# 训练

model.train()

loss\_total = 0

st = time.time()

# train\_dataloader 加载数据集

for data, label in tqdm\_notebook(train\_dataloader):

output = model(data)

# 计算损失

loss = model.losser(output, label)

model.optimizer.zero\_grad()

# 反向传播

loss.backward()

model.optimizer.step()

loss\_total += loss.item()

# 输出损失、训练时间等

print('epoch {}/{}:'.format(epoch, config.num\_epochs))

print('training loss: {}, time resumed {}s'.format(

loss\_total/\_train\_len, time.time()-st))

# 验证

model.eval()

loss\_total = 0

st = time.time()

correct = 0

for data, label in val\_dataloader:

output = model(data)

loss = model.losser(output, label)

loss\_total += loss.item()

\_, predicted = torch.max(output.data, 1)

correct += (predicted == label).sum().item()

# 如果准确度取得最高，则保存准确度最高的模型

if correct/\_val\_len > best\_accuracy:

best\_model = copy.deepcopy(model)

print('val loss: {}, time resumed {}s, accuracy: {}'.format(

loss\_total/\_val\_len, time.time()-st, correct/\_val\_len))

if not os.path.exists(config.save\_dir):

os.makedirs(config.save\_dir)

# 模型保存

with open(os.path.join(config.save\_dir,'best\_nn\_model.pkl'), 'wb') as file:

save = {

'tfidfVectorizer' : vectorizer,

'nn\_model' : best\_model

}

pickle.dump(save, file)

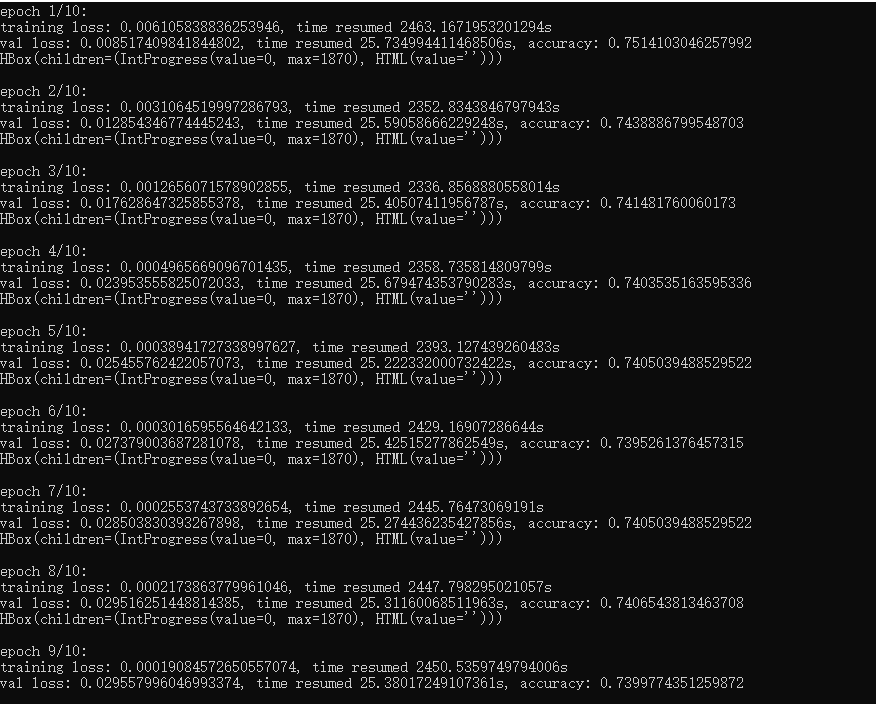


图4 text\_nn训练结果

* 朴素贝叶斯

对于传统的机器学习方法，本模型使用5折交叉检验。

#交叉验证的准确率

cross\_result=cross\_val\_score(pipe,x\_train.cutted\_comment,y\_train,cv=5,scoring='accuracy').mean()

print('交叉验证的准确率：'+str(cross\_result))

1. 模型测试

* TextCNN

加载训练好的模型，对测试集进行预测。利用metrics 输出真实结果和预测值的混淆矩阵，P、R、F1值。代码如下：

# 模型测试

def test(word\_dict,model):

print('加载test语料库=========================')

start\_time = time.time()

test\_df = pd.read\_csv(config.test\_path, sep='\t', header=None, names=["score", "content"], encoding='utf-8')

x\_test, y\_test = build\_word\_dataset(

test\_df, word\_dict, config.stopwords\_path, max\_sen\_len=config.max\_sen\_len, num\_class=config.num\_classes)

with tf.Session() as sess:

init\_op = tf.global\_variables\_initializer()

sess.run(init\_op)

saver = tf.train.Saver()

ckpt = tf.train.get\_checkpoint\_state(config.save\_dir)

print('加载model==========================')

if ckpt and ckpt.model\_checkpoint\_path:

saver.restore(sess, ckpt.model\_checkpoint\_path)

print('Testing============================')

loss\_test, acc\_test = model.evaluate(sess, x\_test, y\_test)

msg = 'Test Loss: {0:>6.4}, Test Acc: {1:>7.4%}'

print(msg.format(loss\_test, acc\_test))

# 预测x

y\_pred = model.predict(sess, x\_test)

y\_cls = np.argmax(y\_test, 1)

# 评估

print("Precision, Recall and F1-Score=====")

print(metrics.classification\_report(y\_cls, y\_pred, target\_names=categories))

# 混淆矩阵

print("Confusion Matrix...")

cm = metrics.confusion\_matrix(y\_cls, y\_pred)

print(cm)

time\_dif = get\_time\_dif(start\_time)

print("Time usage:", time\_dif)

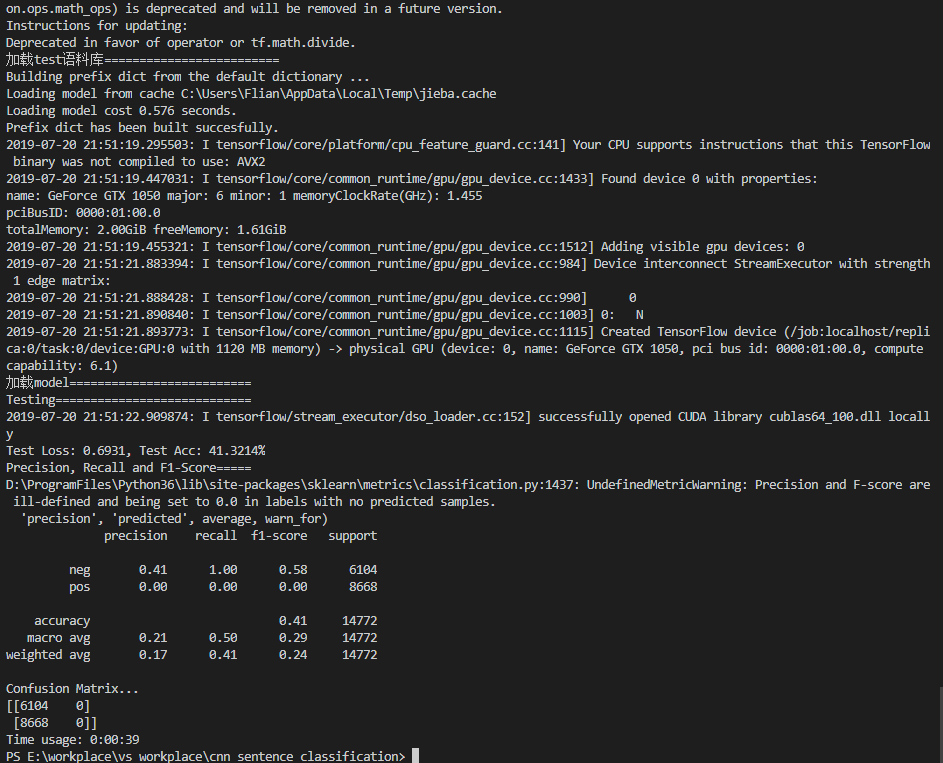


图5 textCNN-5测试结果

* 基于TF-IDF的神经网络

加载训练好的模型，对测试集进行预测。利用metrics 输出真实结果和预测值的混淆矩阵，P、R、F1值。

def nn\_test():

# 加载模型

with open(os.path.join(config.save\_dir,'best\_nn\_model.pkl'), 'rb') as file:

tfidf\_model = pickle.load(file)

tfidfVectorizer = tfidf\_model['tfidfVectorizer']

nn\_model = tfidf\_model['nn\_model']

print('加载test语料库=========================')

start\_time = time.time()

test\_df = pd.read\_csv(config.test\_path, sep='\t', header=None, names=["score", "content"], encoding='utf-8')

x\_test, y\_test = load\_tfidvec\_dataset(test\_df, config.stopwords\_path, tfidfVectorizer)

test\_dataloader, \_test\_len = nn\_model.dataloader(x\_test,y\_test,config.batch\_size,shuffle=False,num\_workers=1)

print('test语料库=========================')

nn\_model.eval()

all\_pred = []

all\_label = []

for data, label in test\_dataloader:

# 评估

\_, predicted = torch.max(nn\_model(data).data, 1)

all\_pred += predicted.numpy().tolist()

all\_label += label.numpy().tolist()

msg = 'Test Acc: {0:>7.4%}'

print(msg.format(sum(all\_pred==all\_label)/\_test\_len))

print("Precision, Recall and F1-Score=====")

print(metrics.classification\_report(all\_label, all\_pred, target\_names=categories))

# 混淆矩阵

print("Confusion Matrix...")

print(metrics.confusion\_matrix(all\_label, all\_pred))

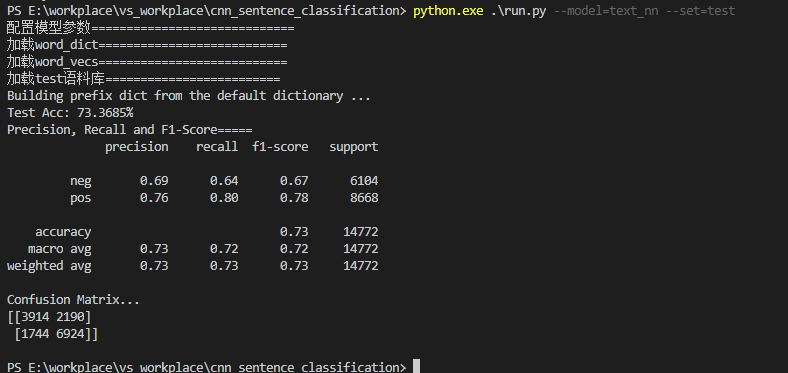


图5 text\_nn测试结果

* 朴素贝叶斯

加载训练好的模型，对测试集进行预测。利用metrics 输出真实结果和预测值的混淆矩阵，P、R、F1值。

#进行预测

pipe.fit(x\_train.cutted\_comment,y\_train)

y\_pred = pipe.predict(x\_test.cutted\_comment)

# 评估

print("Precision, Recall and F1-Score=====")

print(metrics.classification\_report(y\_test, y\_pred, target\_names=categories))

# 混淆矩阵

print("Confusion Matrix...")

cm = metrics.confusion\_matrix(y\_test, y\_pred)

print(cm)

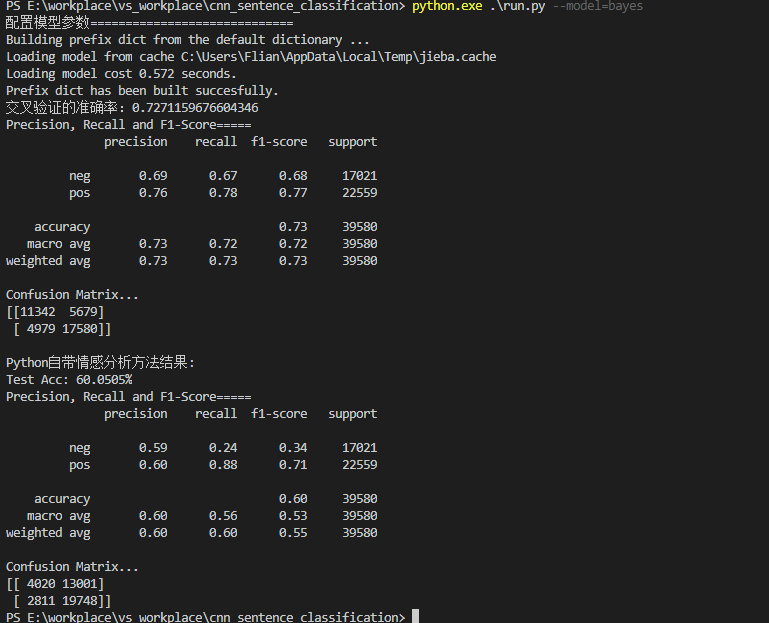


图6 朴素贝叶斯测试结果

1. 实验分析与总结

本实验爬取的语料库，在数据清洗后数据集分布如下：

Data columns (total 2 columns):

score 197900 non-null object

content 197900 non-null object

memory usage: 3.0+ MB

评分0所占比例 0.040051

评分1所占比例 0.112936

评分2所占比例 0.238605

评分3所占比例 0.281885

评分4所占比例 0.326523

可以看出评分0和评分1所占比例很少，评分2、3、4所占比例很大。为了使正负样本比例相近，将0、1、2化为正向，3、4为负向。

在数据过滤后(去掉词数少于5的行)，数据分布如下：

Int64Index: 147713 entries, 0 to 197898

Data columns (total 2 columns):

score 147713 non-null object

content 147713 non-null object

memory usage: 3.4+ MB

0 0.035251

1 0.105915

4 0.244535

2 0.275047

3 0.339252

数据集正负样本比例分布如下：

train: 0.5827440950804874

val: 0.5898458066942459

test: 0.586785810993772

平均句子长度：ave\_sen\_len:11.475786904702204

最大句子长度： max\_sen\_len:869

词总数：vocab\_size:145346

可以看出正负样本比例还是很相近的，可以用于模型训练。但是句子最大长度为869，而平均句子长度为11.48，说明句子长度分布十分不均匀，这导致了TextCNN模型训练结果十分不理想。

故尝试调整句子长度：

* 语料库只保留词数在[5,75]的句子

Int64Index: 145318 entries, 0 to 197898

Data columns (total 2 columns):

score 145318 non-null object

content 145318 non-null object

memory usage: 3.3+ MB

0 0.035226

1 0.105355

4 0.245365

2 0.275190

3 0.338864

train: 0.5833382891416823

val: 0.5939291994800826

test: 0.5827140104596752

>>> ave\_sen\_len:10.82224931397453

>>> max\_sen\_len:75

>>> vocab\_size:139457

* 使用TF-IDF方法，删除超过max\_df=0.95比例的文档中出现的关键词（过于平凡）和低于min\_df=2数量的文档中出现的关键词（过于独特）

整个实验测试结果如下：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 方法 | TextCNN-5 | TextCNN-5\_75 | TextCNN-5\_75  word2vec | TextNN | Bayes | SnowNLP  (可能不太恰当) |
| Train\_acc | 58.9846% | 59.3929% | 59.4159% | 75.1410% |  |  |
| Test\_acc | 41.3214% | 41.7286% | 58.2852% | 73.3685% | 72.7116% | 60.0505% |

改进方向：

* 否定词影响：如”不是很好”分词成了“不是”，“很好”
* 评价内容与评分关联不大。如评价很好，评分却很低；或者评价很差，评分却很高。导致与正常评价冲突，特征混乱。建议考虑每条评论的点赞数
* 调整评价内容长度分布
* 控制词典长度和词选择