案例一：国家电网智能投诉分类

1.案例目的

• 掌握word2vec+lr(逻辑回归)模型的应用；

• 掌握文本预处理方法；

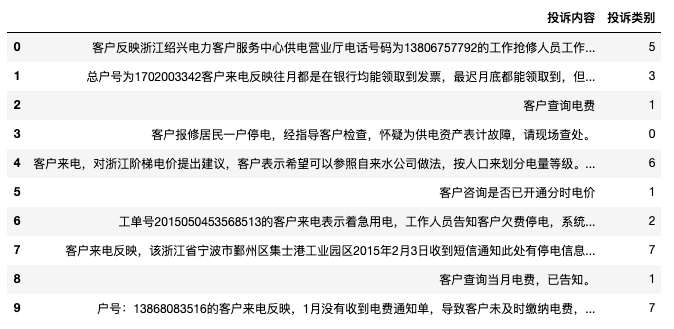
• 掌握使用pytorch深度学习框架搭建TextRCNN模型进行文本分类。

• 掌握文本分类模型的评估

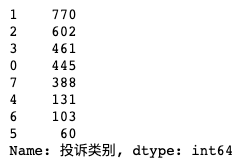
2.案例内容

随着信息化的快速发展，电网各企业积累的数据越来越多。在电力操作规则，运维记录，调度运行记录，营销档案，客服，故障检修等存在着大量文本类数据[由于中文文本类数据的复杂性，无法直接运用机器学习对文本数据进行智能化分析，因此基于电力文本的文本挖掘技术研究显得十分重要。研究电力文本特征提取，电网本体建模。构建调控、检修和营销等知识图谱。支持电力调度机器人，电网设备智能运维和智能客服等，最终形成新一代电力智能搜索和问答解决方案。

本案例主要基于95598工单完成文本分类，选取某省95598工单投诉数据集4229条。其中的工单可以分类为反映、建议、表扬、举报、投诉、以往业务、查询、报修8个类别。数据如下图所示:



其中每一类投诉的文本内容数量为:



可以看到共有8中不同的文本类别,其中下标为5(以往业务)仅有60条文本是数据，属于少类样本。我们的任务就是构建文本分类算法模型，根据用户的投诉文本类容准确判定投诉的类别。

本案例使用2种模型对国家电网投诉做分类，这2种模型分别是word2vec+lr(逻辑回归)模型和TextRCNN模型。

1） 使用word2vec+lr(逻辑回归)模型对95588投诉工单进行分类。

使用gensim工具包基于在搜狐新闻数据集上预训练的word2vec向量进行增量训练，词向量维度为100，分类模型选择逻辑回归，最终结果如图1所示。

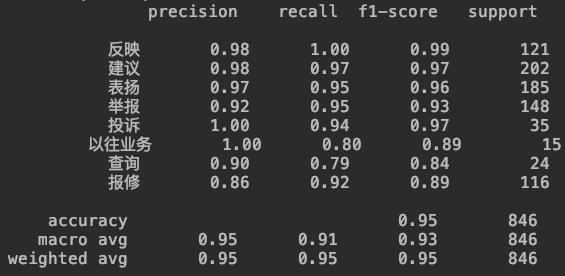


图1 word2vec+lr模型分类结果

从上图中可以看出，word2vec+lr最终在测试集上的准确率为0.95，平均F1值为0.95，效果非常不错，但是依然还有很大的改进空间。

2）使用TextRCNN模型对95588投诉工单进行分类。

利用pytorch深度学习框架搭建TextRCNN模型(2层Bilstm)训练15个epochs，最终效果分别如图2、图3所示。

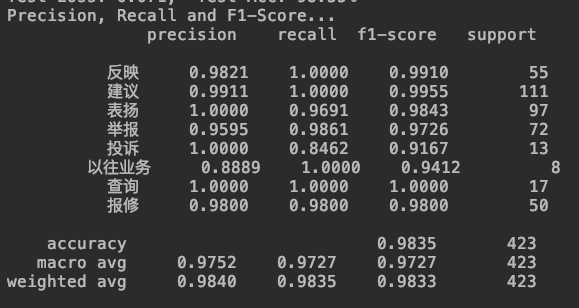


图2 TextRCNN模型分类报告

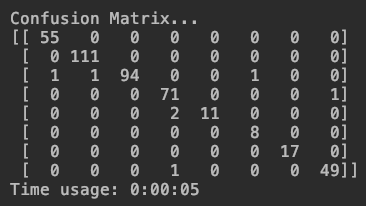


图3 TextRCNN模型混淆矩阵

从上面的两张图中可以看出，相比word2vec+lr模型，TextRCNN模型无论是准确率还是F1值都有了非常大的提高，准确率为0.9835，提高了3.35个百分点，F1值为0.9833，也提高了3个百分点，这说明在该数据集上TextRCNN模型要显著的优于word2vec+lr模型。

3.案例知识点

• gensim工具包使用；

• word2vec增量训练；

• 逻辑回归建模；

• 文本分词和词表构建；

• TextRCNN模型搭建

• 使用pytorch进行模型训练。

4.案例时长

共4学时，学时分配如下：

• word2vec+lr(逻辑回归)模型投诉分类（1学时）；

• TextRCNN模型投诉分类（3学时）。

5.案例实验环境

• Windows 10 操作系统；

• Pycharm

• Python3.6

• pytorch 1.6.0

• tqdm

• sklearn

• genism

• tensorboardX

• jieba

6.案例分析

1）数据介绍和项目准备

• 数据介绍

• 创建项目并拷贝数据文件

2）word2vec+lr(逻辑回归)模型投诉分类

• 实现word2vec增量训练；

• 实现word2vec+LR模型的训练和评估；

3）TextRCNN模型投诉分类

• 实现文本预处理；

• 实现数据集加载和批次迭代；

• 搭建TextRCNN模型；

• 搭建模型训练和测试流程;

• 实现各环节总装并完成训练和评估;

7.案例实验过程

7.1数据介绍和项目准备

1) 创建项目

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

2) 创建“95588”目录

在TextRcnn-Classification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。

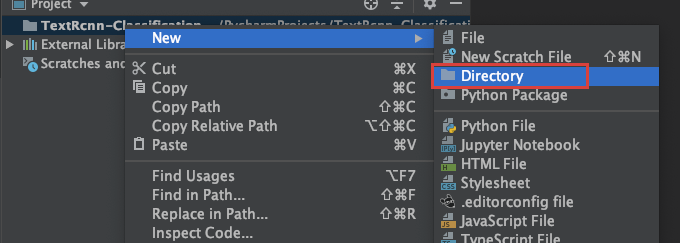


图4 创建目录菜单

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“95588”；

4）点击“OK”按钮。

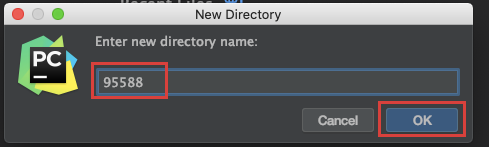


图5 创建目录向导

3) 创建其他目录

    按照同样的方法创建目录“models”，并在“95588”目录下创建“data”、“log”、“save\_dict”和“vectors”等目录，最终结果如图6所示：

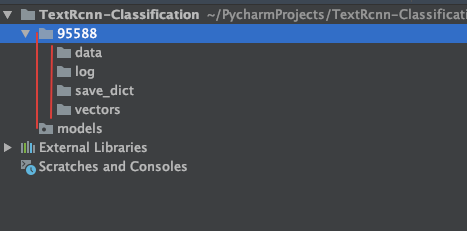


图6 创建目录结果

4) 复制文件

将案例提供的数据文件复制到“95588-data”目录下，结构如图7所示：

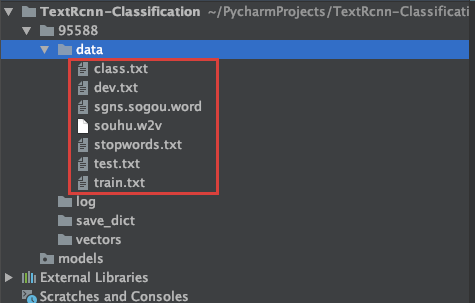
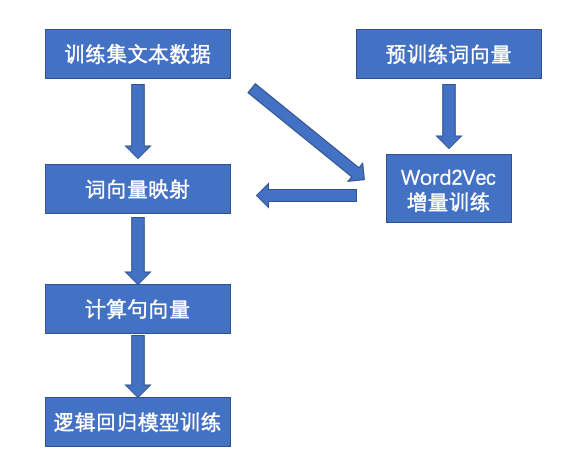


图7 复制数据文件结果

7.2word2vec+lr(逻辑回归)模型投诉分类

本案例中，我们将采用传统机器学习模型和深度学习模型两种方式搭建文本分类预测模型，并对两种模型的结果进行对比。首先我们采用传统的机器学习模型逻辑回归来搭建我们的文本分类预测模型。该模型架构如下图所示：



• 第一步：使用gensim工具包加载预训练词向量

• 第二步：加载训练集文本数据，使用gensim进行word2vec增量训练，得到新的词向量

• 第三步：对训练集文本进行分词

• 第四步：使用训练的词向量对词进行嵌入映射得到文本中每个词对应的词向量

• 第五步：根据文本词向量求平均值，计算文本句向量

• 第六步：送入逻辑回归模型进行训练

下面我们分两部分进行讲解：

• 实现word2vec增量训练；

• 实现word2vec+LR模型的训练和评估；

为了能够让预训练的词向量中包含本项目训练集文本中的语义信息，我们需要在预训练词向量的基础上对训练集文本进行增量训练。为了方便实现训练，我们定义了一个类TrainWord2Vec，该类主要有以下几个方法：

• \_\_init\_\_：初始化方法，该方法主要用来加载停用词以及对模型中的参数：词向量长度、最低词频，窗口大小、是否增量训练和预训练词向量文件等进行配置；

• \_get\_stopwords：该方法用来加载通用词；

• get\_text：该方法负责文本预处理，即对文本进行分词并去除停用词；

• get\_model：该方法负责重新训练word2vec；

• update\_model：该方法负责增量训练word2vec；

• run：该方法负责实现整个训练流程；

首先，在TextRcnn-Classification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：

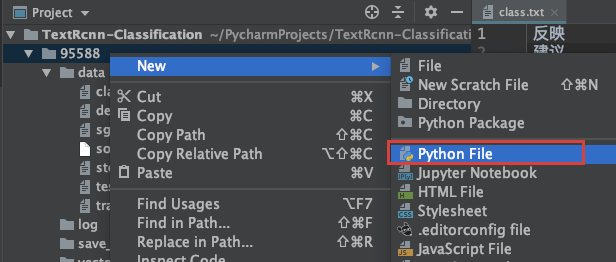


图8 新建py文件菜单

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“train\_w2v”；

4）点击“OK”按钮。

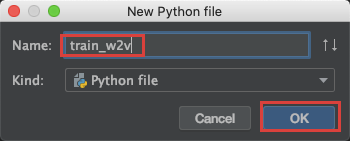


图9 新建py文件向导

在train\_w2v文件中，编写word2vec增量训练代码。

1） 导入使用的包。

这里我们导入了两个工具:

• genism中的Word2Vec类,后面我们将用它来训练word2vec词向量;

• jieba,这是一个用来对中文进行分词的工具包，可以将一个中文句子切分成一个个的词；

【代码1】train\_w2v.py

#导入使用的包

from gensim.models.word2vec import Word2Vec

import jieba

2） 编写TrainWord2Vec类及其\_\_init\_\_方法代码。

在\_\_init\_\_方法中，我们主要进行了以下工作：

• 首先，我们调用了一个加载停用词的方法（该方法将在下一步中实现），把停用词加载进来，因为我们对文本进行分词后需要去掉停用词（一般指对分类没有意义的常用词）；

• 然后，我们对几个重要的属性进行赋值，分别为：

• 训练词向量的长度num\_features：就是指的最终词向量的维度；

• 最低词频min\_word\_count：对于出现频率小于该值的词，我们直接去掉；

• 滑动窗口大小context：指的就是word2vec算法中上下文的词个数（窗口）；

• 开关incremental：该参数指定是否进行增量训练；

• 预训练模型路径old\_path：如果进行增量训练，需要指定该路径，表示预训练模型的路径；

【代码2】train\_w2v.py

#编写word2vec增量训练类

class TrainWord2Vec:

#初始化方法

def \_\_init\_\_(self, stopwords\_file,

num\_features=50,

min\_word\_count=1,

context=5,

incremental=False,

old\_path=None):

"""

定义变量

:param stopword: 停用词表

:param num\_features: 返回的向量长度

:param min\_word\_count: 最低词频

:param context: 滑动窗口大小

:param incremental: 是否进行增量训练

:param old\_path: 若进行增量训练，原始模型路径

"""

# 加载停用词

self.stopwords=self.\_get\_stopwords(stopwords\_file)

self.num\_features = num\_features

self.min\_word\_count = min\_word\_count

self.context = context

self.incremental = incremental

self.old\_path = old\_path

3） 编写完成加载停用词的\_get\_stopwords方法代码。

该方法负责根据给定的停用词文件路径参数，加载指定的停用词，这些词都是对文本分类没有意义的常用词，在文本预处理是需要去掉，最终加载的停用词以集合set返回。

【代码3】train\_w2v.py

def \_get\_stopwords(self,file\_path):

"""

该方法用来从停用词文件中加载停用词

:param file\_path: 停用词路径

:return: 停用词集合

"""

#创建集合

stopwords = set()

#打开文件并遍历每一行

with open(file\_path, 'r', encoding='utf-8') as infile:

for line in infile:

line = line.rstrip('\n')#去掉右边的换行符

if line:

stopwords.add(line.lower())#转换成小写后添加到集合中

return stopwords

4） 编写完成加载训练数据的get\_text方法代码。

该方法从训练数据中读取文本数据并进行分词，去除停用词；步骤如下:

• 打开文件

• 遍历每一行,对每一行进行分词

• 遍历每一个词,判断是否是停用词,不是停用词就保留

最终返回的是有这些文章构成的列表，形式如：[[词11,词12,…],[词21,词22,…],[ 词31,词32,…],…]

【代码4】train\_w2v.py

def get\_text(self,content\_file):

"""

该方法从训练数据中读取文本数据并进行分词，去除停用词，最终返回分次之后的文章列表

:param file\_path: 训练文件路径 每行结构： 文本 \t 标签

:param stopwords\_file: 停用词文件路径

:return: 由每行文本的词构成的列表的列表

"""

corpus = []

# 打开文本 遍历每一行

for line in open(content\_file, 'r', encoding='utf-8'):

curr\_words=[]

# 遍历每一行中的每个词

for word in jieba.cut(line.strip().split('\t')[0]):

# 去除停用词

if word not in self.stopwords:

curr\_words.append(word)

corpus.append(curr\_words)

print(corpus)

return corpus

5） 编写完成word2vec训练的get\_model方法代码。

该方法负责重新训练模型，其中我们实例化了一个Word2Vec对象并进行模型训练，最后将训练得到的模型返回。其接收了4个参数：

• sentences：经过分词清洗之后的语料；

• size：词向量的维度（长度）；

• min\_count：最小词频，词频少于该值的词将不被使用和训练；

• window：上下文窗口大小；

【代码5】train\_w2v.py

def get\_model(self, text):

"""

从头训练word2vec模型

:param text: 经过清洗之后的语料数据

:return: word2vec模型

"""

model = Word2Vec(sentences=text,

size=self.num\_features,

min\_count=self.min\_word\_count,

window=self.context)

return model

6） 编写完成增量训练的update\_model方法代码。

该方法负责进行增量训练，分为以下几个步骤实现：

• 加载预训练的旧模型；

• 根据新的语料text更新词表，因为新的语料中可能出现之前的模型没有见过的词；

• 进行增量训练

• 将训练之后的模型返回

【代码6】train\_w2v.py

def update\_model(self, text):

"""

增量训练word2vec模型

:param text: 经过清洗之后的新的语料数据

:return: word2vec模型

"""

model = Word2Vec.load(self.old\_path) # 加载旧模型

model.build\_vocab(text, update=True) # 更新词汇表

model.train(sentences=text,

total\_examples=model.corpus\_count,#total\_examples:句子数。

epochs=model.iter) # epoch=iter语料库的迭代次数；（默认为5）

return model

7） 编写完成训练流程的run方法代码。

该方法中我们实现了整个业务的流程，流程如下：

• 调用get\_text对文本进行分词、去停用词等处理，并返回处理后的文章数据；

• 判断是否进行增量训练：

• 是：将上一步处理后的text文本传给update\_model方法进行增量训练；

• 不是：将上一步处理后的text文本传给get\_model方法进行重新训练；

• 保存模型和词向量

至此，整个词向量训练模型类搭建完毕，马上我们就可以编写主函数，调用该类进行训练。

【代码7】train\_w2v.py

def run(self,content\_file,save\_path):

"""

负责完成整个训练和保存的流程

:param content\_file: 训练预料文件路径

:param save\_path: 词向量模型保存路径

:return: None

"""

#提取文本分词结果

text = self.get\_text(content\_file)

if self.incremental:#判断是否增量训练

model = self.update\_model(text)#增量训练

else:

model = self.get\_model(text)#重新训练

# 保存模型

model.save(save\_path+"/word2vec\_new.model")#保存模型

model.wv.save\_word2vec\_format(save\_path+'/word2vec\_format\_new1.txt')#保存词向量

8） 编写main函数实例化TrainWord2Vec对象完成训练。

在main函数中，我们提供了停用词文件路径、训练预料文件路径、模型保存路径和词向量保存路径，然后实例化我们刚刚实现的TrainWord2Vec类的对象，并指定了incremental=True（代表进行增量训练），最后调用run方法完成训练和模型保存。

【代码8】train\_w2v.py

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

stopword='95588/data/stopwords.txt'#停用词文件路径

content\_file = '95588/data/train.txt'#训练语料文件路径

save\_path = '95588/vectors'#word2vec模型文件路径

old\_path = '95588/data/souhu.w2v'#预训练的word2vec模型文件路径

#实例化对象

trainmodel = TrainWord2Vec(incremental=True,

stopwords\_file=stopword,

old\_path=old\_path)

#训练并保存

trainmodel.run(content\_file ,save\_path)

9）运行train\_w2v.py文件，完成词向量训练。在train\_w2v.py文件空白处点击右键，选择“Run ‘train\_w2v’“，运行文件，如图10所示。运行后，到

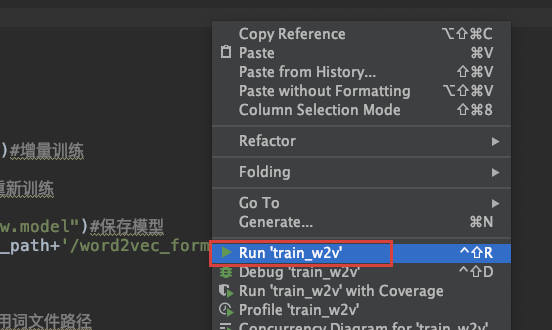


图10 运行train\_w2v

在word2vec词向量训练完成后，我们就可以开始Word2vec+LR模型的搭建了，下面我们实现负责Word2vec+LR模型搭建训练的类LrW2vModel，该类的主要方法有：

• \_\_init\_\_：该方法为初始化方法，主要实现词向量加载、类别标签加载和逻辑回归模型实例化；

• \_load\_w2v：该方法负责加载词向量，在\_\_init\_\_方法中调用

• \_get\_text\_label：该方法负责文本预处理（分词）

• \_compute\_doc\_vec\_single：该方法负责计算一篇文章的句向量

• \_compute\_doc\_vec：该方法调用\_compute\_doc\_vec\_single方法计算所有文章的句向量

• train：该方法实现训练主体流程：文本预处理、文本表示和模型训练

• evaluate：该方法负责模型的评估，最终输出分类报告

首先，在TextRcnn-Classification项目中新建Python File文件“word2vec+lr”。

1）导入需要使用的包。

【代码9】word2vec+lr.py

from gensim.models import Word2Vec

import numpy as np

import jieba#jieba分词

from sklearn.linear\_model import LogisticRegression#逻辑回归

from sklearn.metrics import classification\_report#分类报告函数

2） 编写LrW2vModel类，实现word2vec+lr模型的搭建。首先实现\_\_init\_\_方法

该方法接收两个参数：

• w2v：词向量文件路径，用于加载词向量；

• labels：分类标签文件路径，用于加载类别标签；

在该方法中，我们主要做了三项工作：

• 加载词向量，用于对文本的特征进行计算；

• 加载类别标签，用于显示预测的类别标签

• 初始化逻辑回归，用于后面的训练；

【代码10】word2vec+lr.py

class LrW2vModel(object):

def \_\_init\_\_(self,w2v,labels):

"""

初始化模型

:param w2v: 词向量模型路径

:param labels: 标签文件路径

"""

#加载词向量

self.w2v=self.\_load\_w2v(w2v)

#加载标签

self.class\_list= [x.strip() for x in open(labels, encoding='utf-8').readlines()]

#初始化逻辑回归模型

self.model=LogisticRegression(multi\_class='multinomial', solver='lbfgs')

3） 编写\_load\_w2v方法

该方法负责中调用Word2Vec.load方法，加载预先训练好的词向量模型

【代码11】word2vec+lr.py

#加载词向量模型

def \_load\_w2v(self,path):

w2v = Word2Vec.load(path)

return w2v

4） 编写\_get\_text\_label函数

该方法负责文本的预处理，包含以下几个步骤：

• 遍历文件的每一行，将文本x和标签y分开；

• 对文本x进行分词

• 返回分词后的文本和标签

【代码11】word2vec+lr.py

def \_get\_text\_label(self,file):

"""

获取文本词列表和标签

:param file: 训练文本文件路径

:return: 文本的词列表，文本标签

"""

articles=[]

labels=[]

# 打开文本 遍历每一行

for line in open(file, 'r', encoding='utf-8'):

text,label=line.strip().split('\t')#提取文本内容和标签

articles.append(jieba.lcut(text))#文本分词

labels.append(label)

lens=np.array([len(x) for x in articles])

print("最长:{},最短:{},平均:{}".format(lens.max(),lens.min(),lens.mean()))

return articles,np.array(labels).astype(np.int)

5） 编写\_compute\_doc\_vec\_single方法

该方法负责计算一篇文本的向量表示，分以下几个步骤处理：

• 构建一个词向量长度的全0一维数组；

• 遍历文本每个词，判断是否在词向量中存在；

• 若存在，累加该词的词向量，同时计数器加1

• 计算文本向量=累加向量/词个数

• 返回计算得到的文本向量

这样，我们将每一个文本表示成一个跟词向量等长的向量，代表的是文本中所有词的词向量的平均值。

【代码11】word2vec+lr.py

def \_compute\_doc\_vec\_single(self,article):

"""

返回文本中所有词的词向量的平均值作为文本向量表示

:param article: 一个文本

:return: 返回文本的向量表示

"""

vec = np.zeros((self.w2v.layer1\_size,), dtype=np.float32)

n = 0

for word in article:#遍历文本的每个词

if word in self.w2v:#如果在词表中

vec += self.w2v[word]#累加词向量

n += 1#累加词的个数

return vec / n#求均值

6） 编写\_compute\_doc\_vec方法

该方法遍历每一个文本，调用\_compute\_doc\_vec\_single方法计算每一个文本的向量表示，最终返回数据形式的文本特征矩阵，矩阵形状：[文本数，词向量长度]。

【代码11】word2vec+lr.py

#计算文本的词向量表示

def \_compute\_doc\_vec(self,articles):

#遍历每个文本，计算文本向量表示

return np.row\_stack([self.\_compute\_doc\_vec\_single(x) for x in articles])

7） 编写train方法

该方法调用前面定义的方法和模型完成模型训练流程，主要步骤：

• 处理数据，得到文本和标签；

• 提取向量特征，对文本进行向量表示；

• 逻辑回归模型训练，根据处理好的x和labels，进行模型训练

【代码11】word2vec+lr.py

def train(self,data):

#1.处理数据

articles,labels=self.\_get\_text\_label(data)

#2.计算句向量

x=self.\_compute\_doc\_vec(articles)

#3.模型训练

self.model.fit(x,labels)

8） 编写evaluate方法

该方法负责使用验证集数据来对训练好的模型进行评估，主要步骤为：

• 加载验证集数据并处理；

• 提取文本向量特征；

• 使用训练的模型进行预测；

• 输出分类报告，对模型进行评估

【代码11】word2vec+lr.py

def evaluate(self,data):

#1.处理数据

articles, labels = self.\_get\_text\_label(data)

# 2.计算句向量

x = self.\_compute\_doc\_vec(articles)

#3.模型预测

y\_pred=self.model.predict(x)

#4.模型评估 分类报告

return classification\_report(y\_pred=y\_pred,

y\_true=labels ,

labels=[0,1,2,3,4,5,6,7],

target\_names=self.class\_list)

9） 编写main函数，实例化LrW2vModel对象并进行训练和评估

main函数中，我们制定了训练集文件、验证集文件、词向量文件和标签文件，然后实例化前面实现的模型类，并调用train方法进行训练，最后调用evaluate方法进行评估。

【代码11】word2vec+lr.py

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

train\_data='95588/data/train.txt'

dev\_data = '95588/data/dev.txt'

w2v='95588/vectors/word2vec\_new.model'

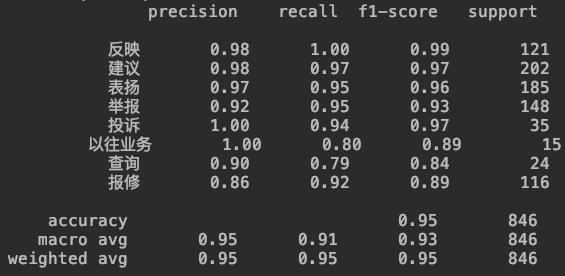
labels='95588/data/class.txt'

model=LrW2vModel(w2v,labels)

model.train(train\_data)

print(model.evaluate(dev\_data))

10）运行**word2vec+lr.py文件**，训练模型并输出分类报告。运行结果如下图所示：



7.3TextRCNN模型投诉分类

上一节我们实现了word2vec+lr模型进行文本分类，为了让分类的效果能够更好一些，本机我们尝试使用基于深度学习的模型TextRCNN来实现投诉分类。我们将分一下几个步骤实现模型搭建：

• 文本预处理：主要工作为：文本分词、词表构建和预训练词向量加载

• 数据集加载和批次迭代：主要工作为：加载训练集数据并封装成可批次迭代的数据结构

• 搭建TextRCNN模型：主要工作为：使用pytorch搭建TextRCNN网络结构并编写配置类congfig

• 搭建模型训练和测试流程：搭建完整的训练流程和测试流程

• 总装并完成训练和测试：负责将各个环节进行总装并运行程序实现训练和测试

文本送入模型训练前需要对文本进行预处理，就是讲文本转换成对应的词向量。其步骤为：

• 对训练集文本进行分词处理

• 提取词频，并抛弃词频过低的词，保留的词就是我们的词表

• 在词表中加入<UNK>来表示未知词（解决未登录词为）

• 在词表中加入<PAD>来表示填充，因为文本序列送入模型前要统一长度，需要对短文本机进行填充

• 使用预训练词向量对构建的词表进行文本表示

在TextRcnn-Classification项目中新建Python File文件“utils”。

1）导入需要使用的包。

【代码12】utils.py

import os

import torch

import numpy as np

import pickle as pkl

from tqdm import tqdm

import time

from datetime import timedelta

import jieba

2） 编写创建词表的函数build\_vocab。

这里我们首先定义三个变量并赋值:

• MAX\_VOCAB\_SIZE：该变量是来控制词表的大小，我们希望将词表大小（非重复词的个数）控制在10000以内，这样可以避免词表太大带来的过拟合问题。

• UNK：我们给该变量赋值'<UNK>'，意思是在训练和预测过程中，没有再词表中出现的词我们都统一转换成'<UNK>'，通过这种方式来处理未登录词问题。

• PAD：赋值为'<PAD>'，代表填充字符。因为句子长短不一，我们需要将句子统一长度，那么就要对短句子进行填充，填充的字符就是'<PAD>'。

然后再函数中我们读取训练文本，并对每一个文本进行分词，统计词频，得到词表。具体步骤如下：

• 打开训练文本文件；

• 遍历每一行，对每一行进行分词，同时统计每个词出现的频率，最终得到所有词的词频数据；

• 过滤词频较低的词并根据词频对词进行降序排序，词频大的在前，小的在后；

• 构建词典，形式为字典形式：{词：对应下标}，后续我们需要用这个词表将每个词转换成id形式；

• 将'<UNK>'和'<PAD>'加入到词表的最后面；

【代码13】utils.py

MAX\_VOCAB\_SIZE = 10000 # 词表长度限制

UNK, PAD= '<UNK>', '<PAD>' # 未知字，padding符号,数字

def build\_vocab(file\_path, tokenizer, max\_size, min\_freq):

"""

通过训练文本进行词表构建

:param file\_path: 训练文件路径

:param tokenizer: 分词器

:param max\_size: 词典最大长度

:param min\_freq: 最小词频

:return:

"""

#空字典，用来保存词表

vocab\_dic = {}

#打开训练文件

with open(file\_path, 'r', encoding='UTF-8') as f:

for line in tqdm(f):#遍历每一行

lin = line.strip()#去除空格

if not lin:

continue

content = lin.split('\t')[0]#提取文本内容

for word in tokenizer(content):#分词

vocab\_dic[word] = vocab\_dic.get(word, 0) + 1#统计词频

#过滤词频较低的词并按照词频排序

vocab\_list = sorted([\_ for \_ in vocab\_dic.items() if \_[1] >= min\_freq],

key=lambda x: x[1],

reverse=True)[:max\_size]

#构建id：word字典

vocab\_dic = {word\_count[0]: idx for idx, word\_count in enumerate(vocab\_list)}

#将UNK和PAD加入词表

vocab\_dic.update({UNK: len(vocab\_dic), PAD: len(vocab\_dic) + 1})

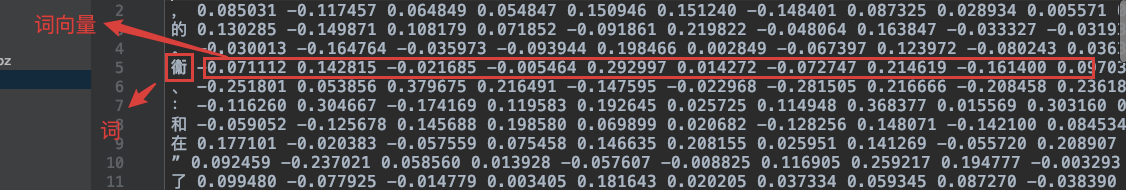
return vocab\_dic

3）编写提取预训练词向量的函数initial\_embedding。

该函数中，我们主要目的是使用预训练的词向量来对我们词表中的词进行向量化，并保存为文件。具体步骤如下：

• 使用np.random.rand生成一个随机数矩阵，形状为【词表大小，词向量维度】，该矩阵的作用是将每个词映射成一个固定长度向量。

• 打开预训练词向量文件，该文件内容如下图所示：



• 遍历文件每一行，读取词和词向量，如果该词在我们的词表中，就对上一步生成的矩阵中该词对应的向量进行更新；

• 将更新后的词向量矩阵保存为文件，以备后续模型训练时使用；

【代码14】utils.py

def initial\_embedding(pretrain\_dir,word\_to\_id,emb\_dim):

"""

根据词表随机初始化词向量，然后使用预训练词向量进行更新

:param word\_to\_id: 预训练词向量文件路径

:param word\_to\_id: 词表字典

:param emb\_dim: 词向量维度

:return:

"""

#随机初始化

embeddings = np.random.rand(len(word\_to\_id), emb\_dim)

f = open(pretrain\_dir, "r", encoding='UTF-8')#打开预训练词向量文件

for i, line in enumerate(f.readlines()):#遍历每一行

lin = line.strip().split(" ")#空格分隔开

if lin[0] in word\_to\_id:#如果预训练词在词表中 就更新

idx = word\_to\_id[lin[0]]

emb = [float(x) for x in lin[1:301]]

embeddings[idx] = np.asarray(emb, dtype='float32')

f.close()

# 保存更新后的词表向量

np.savez\_compressed(filename\_trimmed\_dir, embeddings=embeddings)

4）编写main函数，实现词表构建和预训练词向量提取。

编写好上面的两个函数后，我们就可以提供相关文件的路径来执行整个操作，流程如下：

• 提供相关文件路径；

• 提供分词器jieba.cut；

• 调用build\_vocab函数构建词表；

• 将词表保存为文件

• 调用initial\_embedding函数加载预训练词向量并将最终更新的词向量保存为文件；

【代码15】utils.py

if \_\_name\_\_ == "\_\_main\_\_":

'''提取预训练词向量'''

train\_dir = "./95588/data/train.txt"#训练数据文件

vocab\_dir = "./95588/data/vocab.pkl"#词表文件保存路径

pretrain\_dir = "./95588/data/sgns.sogou.word"#预训练词向量文件

emb\_dim = 300#嵌入维度

#最终的词向量文件保存路径

filename\_trimmed\_dir = "./95588/data/embedding\_SougouNews"

tokenizer = jieba.cut # 以词为单位构建词表 使用结巴分词

word\_to\_id = build\_vocab(train\_dir, #构建词表

tokenizer=tokenizer,

max\_size=MAX\_VOCAB\_SIZE,

min\_freq=1)

# 保存词表

pkl.dump(word\_to\_id, open(vocab\_dir, 'wb'))

#读取预训练词向量并更新词表向量

initial\_embedding(pretrain\_dir,word\_to\_id,emb\_dim)

5）运行**utils.py文件**，生成词表和词向量文件。

深度学习模型训练时，由于计算量巨大，我们不能一次性将所有数据送入模型进行训练，因此我们需要对数据集进行批次迭代管理。主要步骤为：

• 加载词表

• 对文本进行分词处理

• 长度短语规定长度的文本在后面填充<PAD>，长度长于规定长度的文本做截断处理，保证文本长度相等

• 返回词id序列，标签和真实长度作为数据集

• 对数据集进行分批，构建迭代器对象

继续在utils.py文件中编写代码。

1） 编写实现数据集加载的函数build\_dataset。该函数负责加载训练集、验证集和测试集数据。由于该函数代码较长，我们分为三个代码块进行说明。

首先该函数接收一个参数config，这个参数是一个庞大的字典，其中保存的是模型的所有配置参数，将在后面模型搭建的时候实现，这里我们先知道一下即可。下方的几行代码中我们主要目的是加载词表，主要逻辑如下：

• 判断给定的词表文件是否存在；

• 存在，直接加载；

• 不存在，指定分词器，调用之前定义的函数build\_vocab重新构建词表；

【代码16】utils.py

def build\_dataset(config):

#判断是否有词表

if os.path.exists(config.vocab\_path):

#直接加载

vocab = pkl.load(open(config.vocab\_path, 'rb'))

else:

#指定分词器

tokenizer = jieba.lcut

#重新构建

vocab = build\_vocab(config.train\_path,

tokenizer=tokenizer,

max\_size=MAX\_VOCAB\_SIZE,

min\_freq=1)

pkl.dump(vocab, open(config.vocab\_path, 'wb'))

print(f"Vocab size: {len(vocab)}")

然后我们在build\_dataset函数内部定义一个函数load\_dataset，函数接收两个参数：

• path：数据文件路径

• pad\_size：文本填充长度，即每个文本都被处理成这个长度，长截短补

该函数功能是根据给定的数据文件路径加载文本数据得到训练需要的文本特征向量和对应的真实标签。逻辑如下：

• 打开数据文件；

• 遍历每一行文本，对每一行文本执行如下操作：

• 获取content和label

• 对content进行分词

• 计算词的个数并和pad\_size比较，长就截断，短就补'<PAD>'

• 利用词表将每个词转换成对应的id，如果在词表中找不到该词，就用'<UNK>'的id代替该词

• 最终，返回所有文本处理的结果，形式是一个元组构成的列表，形式为（所有词的id，标签，真实长度）

【代码17】utils.py

#加载数据函数

def load\_dataset(path, pad\_size=32):

contents = []

#读取文件

with open(path, 'r', encoding='UTF-8') as f:

#遍历每一行

for line in tqdm(f):

lin = line.strip()

if not lin:

continue

#获取文本内容和标签

content, label = lin.split('\t')

words\_line = []

#文本进行分词

token = tokenizer(content)

#计算文本词个数

seq\_len = len(token)

#如果指定了pad\_size

if pad\_size:

#文本长度小于pad\_size

if len(token) < pad\_size:

#在文本序列后面添加PAD字符

token.extend([PAD] \* (pad\_size - len(token)))

else:

#如果文本长度>pad\_size,就截取前pad\_size个词

token = token[:pad\_size]

seq\_len = pad\_size

# 将词转换成对应的id

for word in token:

words\_line.append(vocab.get(word, vocab.get(UNK)))

#返回每句话的id序列，标签和真实长度

contents.append((words\_line, int(label), seq\_len))

return contents

最后，调用刚刚定义的函数load\_dataset，分别加载并转换训练集、验证集合测试集数据并返回。

最终，整个build\_dataset函数返回值为：词表，训练集数据，验证集数据，测试集数据

【代码18】utils.py

#加载训练集

train = load\_dataset(config.train\_path, config.pad\_size)

#加载验证集

dev = load\_dataset(config.dev\_path, config.pad\_size)

#加载测试集

test = load\_dataset(config.test\_path, config.pad\_size)

return vocab, train, dev, test

2） 编写实现数据批次迭代的类DatasetIterater。该函数负责根据数据集构建一个迭代器，每次吐出一个批次的数据。

首先我们实现DatasetIterater类的\_\_init\_\_初始化方法，该方法接收三个参数：

• batches：所有数据

• batch\_size：批次大小，就是没一小批的数据个数

• device：设备，CPU还是GPU

在该方法中，我们主要完成以下工作：

• 属性赋值，批次大小，数据，索引设备等；

• 计算批数，批数=总数据量/批次大小，注意这里可能不能整除；

• 定义一个属性residue，代表“总数据量/批次大小”是否是整数，是为False，不是为True；

【代码19】utils.py

class DatasetIterater(object):

def \_\_init\_\_(self, batches, batch\_size, device):

"""

根据数据集构建一个迭代器，每次吐出一个批次的数据

:param batches: 数据集

:param batch\_size: 批次大小

:param device: 设备 CPU还是GPU

"""

self.batch\_size = batch\_size

self.batches = batches

#计算总批次

self.n\_batches = len(batches) // batch\_size

self.residue = False # 记录batch数量是否为整数

if len(batches) % self.n\_batches != 0:

self.residue = True

self.index = 0

self.device = device

编写方法\_to\_tensor，该方法功能是将特征x，标签y和序列长度seq\_len定义为torch的tensor。

【代码20】utils.py

def \_to\_tensor(self, datas):

#为x和y定义Tensor

x = torch.LongTensor([\_[0] for \_ in datas]).to(self.device)

y = torch.LongTensor([\_[1] for \_ in datas]).to(self.device)

# pad前的长度(超过pad\_size的设为pad\_size)

seq\_len = torch.LongTensor([\_[2] for \_ in datas]).to(self.device)

return (x, seq\_len), y

编写迭代器迭代方法\_\_next\_\_，该方法负责返回截取的一个批次的数据，其中的逻辑为：

• residue为True（代表数据不能整除批次）且index= n\_batches，表示要返回最后一批（余数）的数据，那么就截取到最后一条；

• index>= n\_batches,表示所有批次数据取完，那么就终止迭代；

• index<n\_batches，表示在中间，那么就截取一个批次大小的数据返回；

【代码21】utils.py

def \_\_next\_\_(self):

#判断批次数不为整数且正好是最后一个批次

if self.residue and self.index == self.n\_batches:

#截取到最后一条数据

batches = self.batches[self.index \* self.batch\_size: len(self.batches)]

self.index += 1

batches = self.\_to\_tensor(batches)

return batches

#index超过了 表示一个轮次完了

elif self.index >= self.n\_batches:

self.index = 0

raise StopIteration

else:

#截取一个批次长度数据

batches = self.batches[self.index \* self.batch\_size: (self.index + 1) \* self.batch\_size]

self.index += 1

batches = self.\_to\_tensor(batches)

return batches

最后，定义两个小方法：

• \_\_iter\_\_：重写的一个方法，返回对象本身self；

• \_\_len\_\_：重写的方法，返回迭代器的长度，其逻辑是：

• residue为真，表示数据不能整除，那么长度=n\_batches+1，因为要加上最后那个不能整除批次

• residue为假，表示能够整除，长度就等于n\_batches

【代码22】utils.py

def \_\_iter\_\_(self):

return self

def \_\_len\_\_(self):

if self.residue:

return self.n\_batches + 1

else:

return self.n\_batches

3）编写两个工具函数build\_iterator（负责创建数据迭代器对象）和get\_time\_dif（负责计算时间差）。

【代码23】utils.py

def build\_iterator(dataset, config):

iter = DatasetIterater(dataset, config.batch\_size, config.device)

return iter

def get\_time\_dif(start\_time):

"""获取已使用时间"""

end\_time = time.time()

time\_dif = end\_time - start\_time

return timedelta(seconds=int(round(time\_dif)))

TextRCNN模型为该项目的核心模块，我们使用pytorch深度学习框架搭建TextRCNN网络结构，该部分我们主要实现两个类：

• Config：负责对TextRCNN模型进行配置

• Model：实现TextRCNN模型的网络结构

在TextRcnn-Classification项目中models目录下新建Python File文件“TextRCNN”。

1）导入需要使用的包。

【代码24】models/ TextRCNN.py

import torch

import torch.nn as nn

import torch.nn.functional as F

import numpy as np

2）定义用来配置参数的配置类Config，该类用来保存配置模型的一些参数。每个参数的作用已经在代码中注释，这里不再做解释。

【代码25】models/ TextRCNN.py

class Config(object):

"""配置参数"""

def \_\_init\_\_(self, dataset, embedding):

self.model\_name = 'TextRCNN'

# 训练集

self.train\_path = dataset + '/data/train.txt'

# 验证集

self.dev\_path = dataset + '/data/dev.txt'

# 测试集

self.test\_path = dataset + '/data/test.txt'

# 类别名单

self.class\_list = [x.strip() for x in open(

dataset + '/data/class.txt', encoding='utf-8').readlines()]

# 词表

self.vocab\_path = dataset + '/data/vocab.pkl'

# 模型保存路径

self.save\_path = dataset + '/saved\_dict/' + self.model\_name + '.ckpt'

#模型训练日志路径

self.log\_path = dataset + '/log/' + self.model\_name

# 预训练词向量

self.embedding\_pretrained = torch.tensor(

np.load(dataset + '/data/' + embedding)["embeddings"].astype('float32'))\

if embedding != 'random' else None

# 设备

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

# 随机失活

self.dropout = 0.5

# 早停 若超过1000batch效果还没提升，则提前结束训练

self.require\_improvement = 1000

# 类别数

self.num\_classes = len(self.class\_list)

# 词表大小，在运行时赋值

self.n\_vocab = 0

# epoch数

self.num\_epochs = 15

# 批次大小

self.batch\_size = 64

# 每句话处理成的长度(短填长切)

self.pad\_size = 100

# 学习率

self.learning\_rate = 1e-3

# 字向量维度, 若使用了预训练词向量，则维度统一

self.embed = self.embedding\_pretrained.size(1)\

if self.embedding\_pretrained is not None else 300

# lstm隐藏层神经元数

self.hidden\_size = 256

# lstm层数

self.num\_layers = 2

3)实现TextRCNN模型。TextRCNN模型的架构图如下图所示：

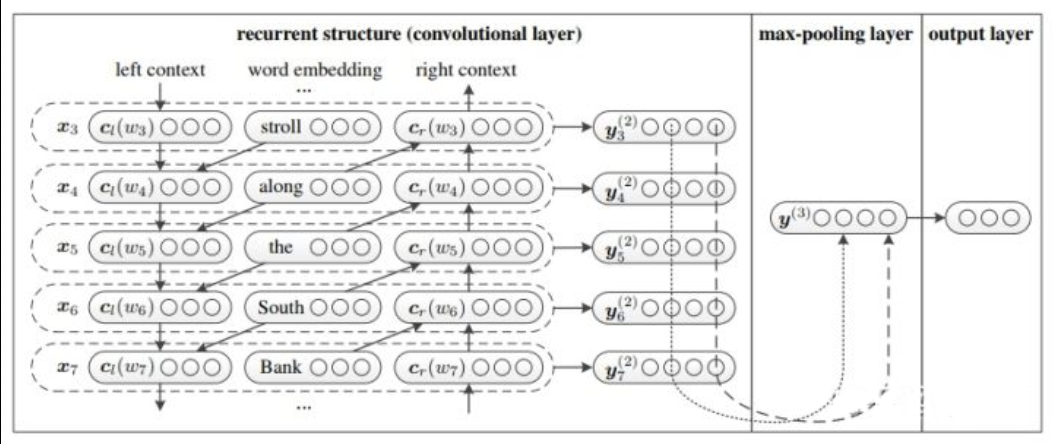


图11 TextRCNN模型

1.模型输入： [batch\_size, seq\_len]

2.经过embedding层：加载预训练词向量或者随机初始化, 词向量维度为embed\_size： [batch\_size, seq\_len, embed\_size]

3.双向LSTM：隐层大小为hidden\_size，得到所有时刻的隐层状态(前向隐层和后向隐层拼接) [batch\_size, seq\_len, hidden\_size \* 2]

4.将embedding层与LSTM输出拼接，并进行非线性激活：

[batch\_size, seq\_len, hidden\_size \* 2 + embed\_size]

5.池化层：seq\_len个特征中取最大的

[batch\_size, hidden\_size \* 2 + embed\_size]

6.全连接后softmax

[batch\_size, num\_class] ==> [batch\_size, 1]

分析：

双向LSTM每一时刻的隐层值(前向+后向)都可以表示当前词的前向和后向语义信息，将隐藏值与embedding值拼接来表示一个词；然后用最大池化层来筛选出有用的特征信息。

下面我们使用pytorch来实现TextRCNN模型类Model：

首先，我们来实现Model类的初始化方法\_\_init\_\_，该方法中实现了textRCNN网络中的四个层：

• 嵌入层embedding：将输入的文本词id映射成对应的向量表示；

• 循环神经网络层BiLstm层：对输入的文本序列进行特征提取；

• 最大池化层MaxPool1d：对文本长度进行归一；

• 全连接层Linear：对文本特征进行映射到num\_class（类别数）进行分类；

【代码26】models/ TextRCNN.py

#继承nn.Module

class Model(nn.Module):

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

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

#如果使用预训练词向量

if config.embedding\_pretrained is not None:

#构建Embedding层并从预训练词向量中加载参数

self.embedding= nn.Embedding.from\_pretrained(

config.embedding\_pretrained,

freeze=False)

else:

#构建Embedding层，不加载参数，后面会随机初始化

self.embedding = nn.Embedding(config.n\_vocab, #词表大小

config.embed,#词嵌入维度

padding\_idx=config.n\_vocab - 1)

#搭建BiLstm层

self.lstm = nn.LSTM(config.embed, #输入维度

config.hidden\_size, #隐藏层维度

config.num\_layers,#层数

bidirectional=True, #是否双向

batch\_first=True,

dropout=config.dropout)

#最大池化层

self.maxpool = nn.MaxPool1d(config.pad\_size)

#全连接层 因为是双向 所以成2 然后加上embed 即： 前向+后向+embed

self.fc = nn.Linear(config.hidden\_size \* 2 + config.embed,

config.num\_classes)

接着我们来编写前向传播方法forward，该方法中使用定义好的四个层来完成整个神经网络的前向计算，计算流程如下：

• 将输入x进行embedding嵌入映射，形状由[batch\_size, seq\_len]变换为[batch\_size, seq\_len, embed\_size]，即每个词的id被映射为一个embed\_size长度的向量；

• 将嵌入后的序列送入双向lstm层进行上下文特征提取，结果是两个形状为[batch\_size, seq\_len, hidden\_size]的数组，一个是前向计算的结果，一个后向计算的结果；

• 将嵌入层的结果embed和双向lstm的结果out在维度2（也就是embed\_size维度）进行拼接，拼接后的形状为[batch\_size, seq\_len, hidden\_size \* 2 + embed\_size]，这一步的作用是将嵌入特征和双向lstm提取的特征进行融合；

• 使用relu激活函数对数组进行激活，激活的目的在于进行非线性变换，提高模型的拟合能力；

• 调换维度，将数组由形状[batch\_size, seq\_len, hidden\_size \* 2 + embed\_size]变成[batch\_size, hidden\_size \* 2 + embed\_size, seq\_len]，目的是将序列长度变换到最后一维，下一步要在这个维度上进行最大池化操作；

• 最大池化操作，在数组最后一个维度seq\_len上进行全局池化，这样，seq\_len就变成了1，数组形状变为[batch\_size, 3\*embedding, 1]；

• squeeze消除维度为1的维度，数形状由[batch\_size, hidden\_size \* 2 + embed\_size, 1]变成[batch\_size, hidden\_size \* 2 + embed\_size]；

• 使用全连接映射将数组映射到num\_calss个输出，这样就可以进行softmax分类了，数组形状由[batch\_size, hidden\_size \* 2 + embed\_size]变成[batch\_size, num\_class]，即每个文本被映射成num\_class长度的向量，进行num\_class分类；

【代码27】models/TextRCNN.py

def forward(self, x):

x, \_ = x

embed = self.embedding(x) #嵌入层映射成向量 [batch\_size, seq\_len, embeding]

out, \_ = self.lstm(embed)#lstm层计算 2\*（[batch\_size, seq\_len, embeding]）

out = torch.cat((embed, out), 2)#拼接前向、后向和embed

out = F.relu(out)#jih

out = out.permute(0, 2, 1)#调换维度

out = self.maxpool(out).squeeze()#池化

out = self.fc(out)#全连接

return out

TextRCNN网络实现后，就可以开始搭建训练流程了，但是考虑到训练和测试流程的代码较多，为了项目整体的易读性，我们将训练、验证和测试过程分别进行封装，下面我们将实现4个主要函数，分别为：

• init\_network：该方法负责在训练开始时，对模型中的参数进行初始化

• train：该方法接收模型、训练集迭代器、验证集迭代器和测试集迭代器并完成整个训练和测试过程

• evaluate：该方法负责在验证集上进行验证

• test：该方法负责在测试集上进行评估

在TextRcnn-Classification项目中新建Python File文件“train\_eval”。

1）导入需要使用的包。

【代码28】train\_eval.py

import numpy as np

import torch

import torch.nn as nn

import torch.nn.functional as F

from sklearn import metrics#模型评估

import time

from utils import get\_time\_dif

from tensorboardX import SummaryWriter#记录训练过程

2）编写初始化权重函数init\_network，该函数负责在构建TextRCNN模型后对模型的参数进行初始化。主要参数如下：

• model：需要进行参数初始化的模型；

• method：参数初始化的方法，就是用什么初始化器对参数进行初始值赋值；

• exclude：指定哪些参数不进行初始化；

• seed：初始化一般都是随机数，这个就是设定随机数种子；

该函数中的逻辑如下：

• for循环遍历模型的参数；

• 判断参数名字是不是不在排除（exclude）的参数中；

• 判断method，根据不同的method设置不同的初始化器进行初始化；

【代码29】train\_eval.py

# 权重初始化，默认xavier

def init\_network(model, method='xavier', exclude='embedding', seed=123):

#遍历模型中的每一个参数

for name, w in model.named\_parameters():

#排除embedding参数，这个加载预训练词向量

if exclude not in name:

#如果是w参数

if 'weight' in name:

if method == 'xavier':

nn.init.xavier\_normal\_(w)

elif method == 'kaiming':

nn.init.kaiming\_normal\_(w)

else:

nn.init.normal\_(w)

#如果是b参数

elif 'bias' in name:

nn.init.constant\_(w, 0)

else:

pass

3） 编写执行训练流程的函数train。该函数负责训练，并每隔100次迭代在验证集上进行验证，所有的epochs走完后在测试集上进行评估。由于该函数代码较长，我们分几个部分进行讲解。

第一部分，我们先对训练过程中需要使用的一些变量和工具进行初始化赋值，主要步骤如下：

• 指定模型为训练（train）模式，在该模式下，模型会自动计算梯度并更新参数；

• 指定优化器optimizer，作用是在训练的过程中对参数进行更新，这里我们使用的是Adam优化器；

• 指定学习率衰减方式scheduler，这里我们使用的是指数衰减，学习率会随着迭代的步数进行指数衰减，不断变小，其目的是为了能够找到一个更好的解；

• 指定summary记录器writer，其作用是对训练过程进行记录，便于对训练过程进行监控；

【代码30】train\_eval.py

def train(config, model, train\_iter, dev\_iter, test\_iter):

start\_time = time.time()

#指定是模型训练模式

model.train()

#实例化优化器

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

# 学习率指数衰减，每次epoch：学习率 = gamma \* 学习率

scheduler = torch.optim.lr\_scheduler.ExponentialLR(optimizer, gamma=0.9)

total\_batch = 0 # 记录进行到多少batch

dev\_best\_loss = float('inf')

last\_improve = 0 # 记录上次验证集loss下降的batch数

flag = False # 记录是否很久没有效果提升

writer = SummaryWriter(log\_dir=config.log\_path + '/' + time.strftime('%m-%d\_%H.%M', time.localtime()))

第二部分，我们主要实现模型训练，主要步骤如下：

• for循环num\_epochs，即将训练集重复迭代num\_epochs次；

• 每次循环进来，首先让学习率衰减一次

• 然后遍历训练集的所有批次数据，对每一个批次的数据作如下几步计算：

• 计算模型输出值

• 梯度清零，准备计算本批次的梯度

• 计算loss损失函数的值

• 损失函数反向传播计算梯度

• 优化器根据梯度更新参数，完成一次迭代

【代码31】train\_eval.py

#遍历num\_epochs次

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

print('Epoch [{}/{}]'.format(epoch + 1, config.num\_epochs))

scheduler.step() # 学习率衰减

#遍历训练集

for i, (trains, labels) in enumerate(train\_iter):

#模型计算输出值

outputs = model(trains)

#梯度清零

model.zero\_grad()

#计算损失

loss = F.cross\_entropy(outputs, labels)

#反相传播计算梯度

loss.backward()

#优化器修改参数

optimizer.step()

第三部分，这一部分我们主要完成模型在验证机上的效果检验，通过观察模型在验证集上的效果，我们能够更好的了解模型优化的过程，并可以通过早停技术来防止模型过度训练导致过拟合。主要的步骤如下：

• 计算模型在训练机上的准确率；

• 调用evaluate函数（后面马上实现）使用验证集评估模型效果，计算出验证集准确率和损失值；

• 判断验证集损失值（dev\_loss）是否小于历史验证集损失值最小值（dev\_best\_loss），如果小于，就保存模型并更新dev\_best\_loss和last\_improve；这样做的目的是保存在验证集行最好的模型；

• 打印训练集和验证集评估数据并写入summary（此步骤代码较多但非核心代码）

【代码32】train\_eval.py

# 每多少轮输出在训练集和验证集上的效果

if total\_batch % 100 == 0:

#真实值

true = labels.data.cpu()

#预测

predic = torch.max(outputs.data, 1)[1].cpu()

#计算准确率

train\_acc = metrics.accuracy\_score(true, predic)

#计算验证集的准确率和损失

dev\_acc, dev\_loss = evaluate(config, model, dev\_iter)

if dev\_loss < dev\_best\_loss:#如果本轮loss小于历史最小loss

dev\_best\_loss = dev\_loss

#保存模型

torch.save(model.state\_dict(), config.save\_path)

improve = '\*'

last\_improve = total\_batch

else:

improve = ''

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

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}'

print(msg.format(total\_batch,

loss.item(),

train\_acc,

dev\_loss,

dev\_acc,

time\_dif,

improve))

#将训练信息写入日志

writer.add\_scalar("loss/train", loss.item(), total\_batch)

writer.add\_scalar("loss/dev", dev\_loss, total\_batch)

writer.add\_scalar("acc/train", train\_acc, total\_batch)

writer.add\_scalar("acc/dev", dev\_acc, total\_batch)

#重新指定为训练模式 因为在上面评估的时候修改成了验证模式

model.train()

第四部分，在验证集评估之后，我们可以使用早停技术来防止过拟合，其思想是记录每次保存最好模型的步数last\_improve，然后使用当前的总步数total\_batch-last\_improve，如果超过设定的阈值1000次，也就意味着模型在1000次迭代中都没有在验证集上取得更好的效果，就终止训练；

最后，在模型所有的epochs迭代完毕后，在测试集上评估模型。

【代码33】train\_eval.py

total\_batch += 1

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

# 验证集loss超过1000batch没下降，结束训练

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

flag = True

break

if flag:

break

writer.close()

#最后在测试集上评估

test(config, model, test\_iter)

4）编写执行验证集验证的函数evaluate。该函数接收参数有以下四个：

• config：配置参数字典，该参数之前我们已经介绍过

• model：训练的textRCNN模型

• data\_iter：数据集迭代器，负责接收数据

• test：控制时候是测试，是测试我们就输出分类报告

该部分代码较长，我们分两个部分进行讲解。

第一部分，我们完成以下步骤：

• 指定模型为eval模式，即评估模式，这样模型就不会计算梯度和更新参数；

• 遍历数据的所有批次数据，对每一个批次我们做以下操作：

• 计算模型输出值

• 计算损失并加入总损失列表中，后续用来计算总损失值

• 计算预测值，并将预测值和真实值加入列表，后续用来评估整个数据集的结果

• 遍历完所有数据后，计算准确率

【代码34】train\_eval.py

def evaluate(config, model, data\_iter, test=False):

"""

评估模型

:param config:配置

:param model: 模型

:param data\_iter: 数据集迭代器

:param test: 是否测试

:return:

"""

# 指定模型为验证模式

model.eval()

loss\_total = 0

predict\_all = np.array([], dtype=int)

labels\_all = np.array([], dtype=int)

with torch.no\_grad():#不计算梯度

for texts, labels in data\_iter:#遍历数据集

outputs = model(texts)#计算输出值

#计算损失

loss = F.cross\_entropy(outputs, labels)

loss\_total += loss

labels = labels.data.cpu().numpy()

#计算预测值

predic = torch.max(outputs.data, 1)[1].cpu().numpy()

labels\_all = np.append(labels\_all, labels)

predict\_all = np.append(predict\_all, predic)

#计算准确率

acc = metrics.accuracy\_score(labels\_all, predict\_all)

第二部分，数据集评估完成后，我们判断是否是测试集，如果是测试集，我们就输出分类报告（classification\_report）和混淆矩阵（confusion），最终返回准确率、平均损失、分类报告（测试才返回）和混淆矩阵（测试才返回）。

【代码35】train\_eval.py

if test:#如果是test测试

#输出分类报告

report = metrics.classification\_report(labels\_all,

predict\_all,

target\_names=config.class\_list,

digits=4)

#输出混淆矩阵

confusion = metrics.confusion\_matrix(labels\_all, predict\_all)

return acc, loss\_total / len(data\_iter), report, confusion

return acc, loss\_total / len(data\_iter)

5）编写执行测试集评估的函数test。该函数接收参数有以下四个：

• config：配置参数字典，该参数之前我们已经介绍过

• model：训练的textRCNN模型

• data\_iter：数据集迭代器，负责接收数据

该函数主要作用是在训练过程完成后，使用测试集数据对训练的模型进行评估。主要流程为：

• 加载模型

• 指定模型为验证模式

• 调用evaluate函数计算准确率、损失、分类报告和混淆矩阵

• 输出信息

【代码36】train\_eval.py

def test(config, model, test\_iter):

# 加载模型

model.load\_state\_dict(torch.load(config.save\_path))

# 指定模型为验证模式

model.eval()

start\_time = time.time()

#测试集评估

test\_acc, test\_loss, test\_report, test\_confusion = evaluate(config, model, test\_iter, test=True)

msg = 'Test Loss: {0:>5.2}, Test Acc: {1:>6.2%}'

print(msg.format(test\_loss, test\_acc))

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

print(test\_report)

print("Confusion Matrix...")

print(test\_confusion)

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

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

在TextRcnn-Classification项目中新建Python File文件“run”。

1）导入需要使用的包。

【代码37】run.py

import time

import torch

import numpy as np

from train\_eval import train, init\_network

from importlib import import\_module

import argparse

from utils import build\_dataset, build\_iterator, get\_time\_dif

2）构建ArgumentParser参数对象。该对象可以接收命令行参数，这里我们设定了一个参数：

• --embedding：该参数的意义是是否使用预训练的词向量

【代码38】run.py

#定义ArgumentParser参数对象

parser = argparse.ArgumentParser(description='Chinese Text Classification')

#指定是否使用预训练词向量

parser.add\_argument('--embedding', default='pre\_trained', type=str, help='random or pre\_trained')

args = parser.parse\_args()

3）写main函数，实现训练。这个函数中，我们使用之前编写的类和函数来实现完整的模型训练流程，下面我们详细介绍一下整个环节：

• 指定数据集名称为95588，这样模型就会去访问对应文件夹的数据；

• 指定预训练词向量文件名为embedding\_SougouNews.npz

• 导入TextRCNN文件为x，并调用该文件的Config类生成参数配置对象

• 调用build\_dataset函数加载词表、训练集、验证集和测试集

• 构建训练集、验证集、测试集迭代器，为训练做数据准备

• 利用参数配置对象config配置并实例化model对象

• 调用之前实现的train函数完成模型训练和评估

【代码39】run.py

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

dataset = '95588' # 数据集

# 搜狗新闻:embedding\_SougouNews.npz, 随机初始化:random

embedding = 'embedding\_SougouNews.npz'

if args.embedding == 'random':

embedding = 'random'

#加载TextRCNN类

x = import\_module('models.TextRCNN')

#生成配置对象

config = x.Config(dataset, embedding)

np.random.seed(1)

torch.manual\_seed(1)

torch.cuda.manual\_seed\_all(1)

torch.backends.cudnn.deterministic = True # 保证每次结果一样

start\_time = time.time()

print("Loading data...")

#加载数据集 训练集、验证集、测试集

vocab, train\_data, dev\_data, test\_data = build\_dataset(config)

#构建训练集、验证集、测试集迭代器

train\_iter = build\_iterator(train\_data, config)

dev\_iter = build\_iterator(dev\_data, config)

test\_iter = build\_iterator(test\_data, config)

#获取时间差

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

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

# train

#加载词表

config.n\_vocab = len(vocab)

#利用配置参数实例化模型

model = x.Model(config).to(config.device)

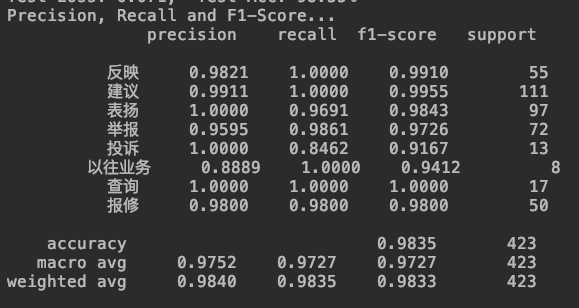
#打印模型参数名

print(model.parameters)

#调用训练函数完成训练

train(config, model, train\_iter, dev\_iter, test\_iter)

4）运行run.py文件，模型开始训练，模型训练完毕后会输出如下图所示的分类报告结果：



8.案例实验结果（结论）

1）word2vec+lr(逻辑回归)模型投诉分类。

在本节案例中，我们采用的是word2vec文本表示和传统的机器学习模型逻辑回归搭建的文本分类算法模型，该模型具有结构简单、性能高等优点，分类效果也非常好，基本能够满足一般场景下的文本分类需求。

我们运行train\_w2v.py文件，然后再运行word2vec+lr.py文件，结果如下图所示：

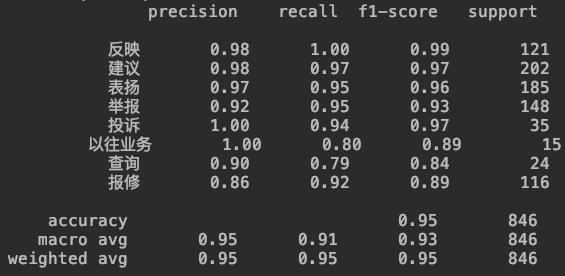


图12 word2vec+lr(逻辑回归)模型投诉分类结果

从结果中可以看到，模型的分类准确率为0.95，具备一般场景下的文本分类能力，但如果是对分类准确率要求较高的场景，该模型还需要提高其准确率。因此我们还需要再尝试使用深度学习模型。

2）使用TextRCNN模型对95588投诉工单进行分类。

为了提高文本分类模型的分类性能，我们尝试使用深度学习模型，这里我们选择了TextRCNN模型，同时为了能够提高模型的泛化能力，我们使用了基于搜狗新闻语料预训练的词向量来对文本进行表示并在本数据集上进行迁移学习（迁移学习是跟TextRCNN模型训练一起完成的，模型训练过程中也对文本表示的词向量进行了修正学习）。

我们先运行utils.py文件，再运行run.py文件，结果如下图所示。

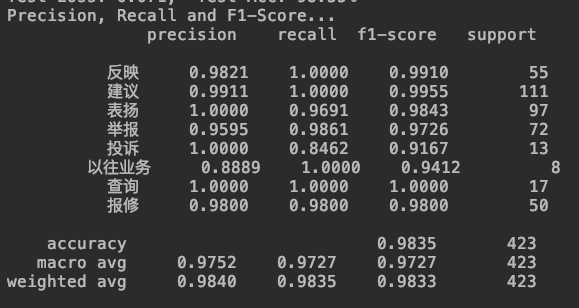


图13 TextRCNN模型分类报告

在上图中国可以看出，TextRCNN模型的分类能力显著高于上个小案例中的逻辑回归模型，这也证明了深度学习模型在文本分类任务中的重要意义。但是，由于深度学习模型较为复杂，计算量较大，其推理速度要低于机器学习的逻辑回归模型，因此，在实际业务场景中，具体使用哪种模型还需要根据具体的情况而定。

9.案例代码

【案例代码】train\_w2v.py

#导入使用的包

from gensim.models.word2vec import Word2Vec

import jieba

#编写word2vec增量训练类

class TrainWord2Vec:

# 初始化方法

def \_\_init\_\_(self, stopwords\_file,

num\_features=50,

min\_word\_count=1,

context=5,

incremental=False,

old\_path=None):

"""

定义变量

:param stopword: 停用词表

:param num\_features: 返回的向量长度

:param min\_word\_count: 最低词频

:param context: 滑动窗口大小

:param incremental: 是否进行增量训练

:param old\_path: 若进行增量训练，原始模型路径

"""

# 加载停用词

self.stopwords=self.\_get\_stopwords(stopwords\_file)

self.num\_features = num\_features

self.min\_word\_count = min\_word\_count

self.context = context

self.incremental = incremental

self.old\_path = old\_path

def get\_text(self,content\_file):

"""

该方法从训练数据中读取文本数据并进行分词，去除停用词，最终返回分次之后的文章列表

:param file\_path: 训练文件路径 每行结构： 文本 \t 标签

:param stopwords\_file: 停用词文件路径

:return: 由每行文本的词构成的列表的列表

"""

corpus = []

# 打开文本 遍历每一行

for line in open(content\_file, 'r', encoding='utf-8'):

curr\_words=[]

# 遍历每一行中的每个词

for word in jieba.cut(line.strip().split('\t')[0]):

# 去除停用词

if word not in self.stopwords:

curr\_words.append(word)

corpus.append(curr\_words)

print(corpus)

return corpus

def \_get\_stopwords(self,file\_path):

"""

该方法用来从停用词文件中加载停用词

:param file\_path: 停用词路径

:return: 停用词集合

"""

#创建集合

stopwords = set()

#打开文件并遍历每一行

with open(file\_path, 'r') as infile:

for line in infile:

line = line.rstrip('\n')#去掉右边的换行符

if line:

stopwords.add(line.lower())#转换成小写后添加到集合中

return stopwords

def get\_model(self, text):

"""

从头训练word2vec模型

:param text: 经过清洗之后的语料数据

:return: word2vec模型

"""

model = Word2Vec(sentences=text,

size=self.num\_features,

min\_count=self.min\_word\_count,

window=self.context)

return model

def update\_model(self, text):

"""

增量训练word2vec模型

:param text: 经过清洗之后的新的语料数据

:return: word2vec模型

"""

model = Word2Vec.load(self.old\_path) # 加载旧模型

model.build\_vocab(text, update=True) # 更新词汇表

model.train(sentences=text,

total\_examples=model.corpus\_count,#total\_examples:句子数。

epochs=model.iter) # epoch=iter语料库的迭代次数；（默认为5）

return model

def run(self,content\_file,save\_path):

"""

负责完成整个训练和保存的流程

:param content\_file: 训练预料文件路径

:param save\_path: 词向量模型保存路径

:return: None

"""

#提取文本分词结果

text = self.get\_text(content\_file)

if self.incremental:#判断是否增量训练

model = self.update\_model(text)#增量训练

else:

model = self.get\_model(text)#重新训练

# 保存模型

model.save(save\_path+"/word2vec\_new.model")#保存模型

model.wv.save\_word2vec\_format(save\_path+'/word2vec\_format\_new1.txt')#保存词向量

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

stopword='95588/data/stopwords.txt'#停用词文件路径

content\_file = '95588/data/train.txt'#训练语料文件路径

save\_path = '95588/vectors'#word2vec模型文件路径

old\_path = '95588/data/souhu.w2v'#预训练的word2vec模型文件路径

#实例化对象

trainmodel = TrainWord2Vec(incremental=True,

stopwords\_file=stopword,

old\_path=old\_path)

#训练并保存

trainmodel.run(content\_file,save\_path)

【案例代码】word2vec+lr.py

from gensim.models import Word2Vec

import numpy as np

import jieba#jieba分词

from sklearn.linear\_model import LogisticRegression#逻辑回归

from sklearn.metrics import classification\_report#分类报告函数

class LrW2vModel(object):

def \_\_init\_\_(self,w2v,labels):

"""

初始化模型

:param w2v: 词向量模型路径

:param labels: 标签文件路径

"""

#加载词向量

self.w2v=self.\_load\_w2v(w2v)

#加载标签

self.class\_list = [x.strip() for x in open(labels, encoding='utf-8').readlines()]

#初始化逻辑回归模型

self.model=LogisticRegression(multi\_class='multinomial', solver='lbfgs')

#加载词向量模型

def \_load\_w2v(self,path):

w2v = Word2Vec.load(path)

return w2v

#计算文本的词向量表示

def \_compute\_doc\_vec(self,articles):

#遍历每个文本，计算文本向量表示

return np.row\_stack([self.\_compute\_doc\_vec\_single(x) for x in articles])

def \_compute\_doc\_vec\_single(self,article):

"""

返回文本中所有词的词向量的平均值作为文本向量表示

:param article: 一个文本

:return: 返回文本的向量表示

"""

vec = np.zeros((self.w2v.layer1\_size,), dtype=np.float32)

n = 0

for word in article:#遍历文本的每个词

if word in self.w2v:#如果在词表中

vec += self.w2v[word]#累加词向量

n += 1#累加词的个数

return vec / n#求均值

def \_get\_text\_label(self,file):

"""

获取文本词列表和标签

:param file: 训练文本文件路径

:return: 文本的词列表，文本标签

"""

articles=[]

labels=[]

# 打开文本 遍历每一行

for line in open(file, 'r', encoding='utf-8'):

text,label=line.strip().split('\t')#提取文本内容和标签

articles.append(jieba.lcut(text))#文本分词

labels.append(label)

lens=np.array([len(x) for x in articles])

print("最长:{},最短:{},平均:{}".format(lens.max(),lens.min(),lens.mean()))

return articles,np.array(labels).astype(np.int)

def train(self,data):

#1.处理数据

articles,labels=self.\_get\_text\_label(data)

#2.计算句向量

x=self.\_compute\_doc\_vec(articles)

#3.模型训练

self.model.fit(x,labels)

def evaluate(self,data):

#1.处理数据

articles, labels = self.\_get\_text\_label(data)

# 2.计算句向量

x = self.\_compute\_doc\_vec(articles)

#3.模型预测

y\_pred=self.model.predict(x)

#4.模型评估 分类报告

return classification\_report(y\_pred=y\_pred,

y\_true=labels ,

labels=[0,1,2,3,4,5,6,7],

target\_names=self.class\_list)

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

train\_data='95588/data/train.txt'

dev\_data = '95588/data/dev.txt'

w2v='95588/vectors/word2vec\_new.model'

labels='95588/data/class.txt'

model=LrW2vModel(w2v,labels)

model.train(train\_data)

print(model.evaluate(dev\_data))

【案例代码】utils.py

import os

import torch

import numpy as np

import pickle as pkl

from tqdm import tqdm

import time

from datetime import timedelta

import jieba

MAX\_VOCAB\_SIZE = 10000 # 词表长度限制

UNK, PAD= '<UNK>', '<PAD>' # 未知字，padding符号,数字

def build\_vocab(file\_path, tokenizer, max\_size, min\_freq):

"""

通过训练文本进行词表构建

:param file\_path: 训练文件路径

:param tokenizer: 分词器

:param max\_size: 词典最大长度

:param min\_freq: 最小词频

:return:

"""

#空字典，用来保存词表

vocab\_dic = {}

#打开训练文件

with open(file\_path, 'r', encoding='UTF-8') as f:

for line in tqdm(f):#遍历每一行

lin = line.strip()#去除空格

if not lin:

continue

content = lin.split('\t')[0]#提取文本内容

for word in tokenizer(content):#分词

vocab\_dic[word] = vocab\_dic.get(word, 0) + 1#统计词频

#过滤词频较低的词并按照词频排序

vocab\_list = sorted([\_ for \_ in vocab\_dic.items() if \_[1] >= min\_freq],

key=lambda x: x[1],

reverse=True)[:max\_size]

#构建id：word字典

vocab\_dic = {word\_count[0]: idx for idx, word\_count in enumerate(vocab\_list)}

#将UNK和PAD加入词表

vocab\_dic.update({UNK: len(vocab\_dic), PAD: len(vocab\_dic) + 1})

return vocab\_dic

def initial\_embedding(pretrain\_dir,word\_to\_id,emb\_dim):

"""

根据词表随机初始化词向量，然后使用预训练词向量进行更新

:param word\_to\_id: 预训练词向量文件路径

:param word\_to\_id: 词表字典

:param emb\_dim: 词向量维度

:return:

"""

#随机初始化

embeddings = np.random.rand(len(word\_to\_id), emb\_dim)

f = open(pretrain\_dir, "r", encoding='UTF-8')#打开预训练词向量文件

for i, line in enumerate(f.readlines()):#遍历每一行

lin = line.strip().split(" ")#空格分隔开

if lin[0] in word\_to\_id:#如果预训练词在词表中 就更新

idx = word\_to\_id[lin[0]]

emb = [float(x) for x in lin[1:301]]

embeddings[idx] = np.asarray(emb, dtype='float32')

f.close()

# 保存更新后的词表向量

np.savez\_compressed(filename\_trimmed\_dir, embeddings=embeddings)

def build\_dataset(config):

#指定分词器

tokenizer = jieba.lcut

#判断是否有词表

if os.path.exists(config.vocab\_path):

#直接加载

vocab = pkl.load(open(config.vocab\_path, 'rb'))

else:

#重新构建

vocab = build\_vocab(config.train\_path,

tokenizer=tokenizer,

max\_size=MAX\_VOCAB\_SIZE,

min\_freq=1)

pkl.dump(vocab, open(config.vocab\_path, 'wb'))

print(f"Vocab size: {len(vocab)}")

#加载数据函数

def load\_dataset(path, pad\_size=32):

contents = []

#读取文件

with open(path, 'r', encoding='UTF-8') as f:

#遍历每一行

for line in tqdm(f):

lin = line.strip()

if not lin:

continue

#获取文本内容和标签

content, label = lin.split('\t')

words\_line = []

#文本进行分词

token = tokenizer(content)

#计算文本词个数

seq\_len = len(token)

#如果指定了pad\_size

if pad\_size:

#文本长度小于pad\_size

if len(token) < pad\_size:

#在文本序列后面添加PAD字符

token.extend([PAD] \* (pad\_size - len(token)))

else:

#如果文本长度>pad\_size,就截取前pad\_size个词

token = token[:pad\_size]

seq\_len = pad\_size

# 将词转换成对应的id

for word in token:

words\_line.append(vocab.get(word, vocab.get(UNK)))

#返回每句话的id序列，标签和真实长度

contents.append((words\_line, int(label), seq\_len))

return contents # [([...], 0), ([...], 1), ...]

#加载训练集

train = load\_dataset(config.train\_path, config.pad\_size)

#加载验证集

dev = load\_dataset(config.dev\_path, config.pad\_size)

#加载测试集

test = load\_dataset(config.test\_path, config.pad\_size)

return vocab, train, dev, test

class DatasetIterater(object):

def \_\_init\_\_(self, batches, batch\_size, device):

"""

根据数据集构建一个迭代器，每次吐出一个批次的数据

:param batches: 数据集

:param batch\_size: 批次大小

:param device: 设备 CPU还是GPU

"""

self.batch\_size = batch\_size

self.batches = batches

#计算总批次

self.n\_batches = len(batches) // batch\_size

self.residue = False # 记录batch数量是否为整数

if len(batches) % self.n\_batches != 0:

self.residue = True

self.index = 0

self.device = device

def \_to\_tensor(self, datas):

#为x和y定义Tensor

x = torch.LongTensor([\_[0] for \_ in datas]).to(self.device)

y = torch.LongTensor([\_[1] for \_ in datas]).to(self.device)

# pad前的长度(超过pad\_size的设为pad\_size)

seq\_len = torch.LongTensor([\_[2] for \_ in datas]).to(self.device)

return (x, seq\_len), y

def \_\_next\_\_(self):

#判断批次数不为整数且正好是最后一个批次

if self.residue and self.index == self.n\_batches:

#截取到最后一条数据

batches = self.batches[self.index \* self.batch\_size: len(self.batches)]

self.index += 1

batches = self.\_to\_tensor(batches)

return batches

#index超过了 表示一个轮次完了

elif self.index >= self.n\_batches:

self.index = 0

raise StopIteration

else:

#截取一个批次长度数据

batches = self.batches[self.index \* self.batch\_size: (self.index + 1) \* self.batch\_size]

self.index += 1

batches = self.\_to\_tensor(batches)

return batches

def \_\_iter\_\_(self):

return self

def \_\_len\_\_(self):

if self.residue:

return self.n\_batches + 1

else:

return self.n\_batches

def build\_iterator(dataset, config):

iter = DatasetIterater(dataset, config.batch\_size, config.device)

return iter

def get\_time\_dif(start\_time):

"""获取已使用时间"""

end\_time = time.time()

time\_dif = end\_time - start\_time

return timedelta(seconds=int(round(time\_dif)))

if \_\_name\_\_ == "\_\_main\_\_":

'''提取预训练词向量'''

train\_dir = "./95588/data/train.txt"#训练数据文件

vocab\_dir = "./95588/data/vocab.pkl"#词表文件保存路径

pretrain\_dir = "./95588/data/sgns.sogou.word"#预训练词向量文件

emb\_dim = 300#嵌入维度

filename\_trimmed\_dir = "./95588/data/embedding\_SougouNews"#最终的词向量文件保存路径

tokenizer = jieba.lcut # 以词为单位构建词表 使用结巴分词

word\_to\_id = build\_vocab(train\_dir, #构建词表

tokenizer=tokenizer,

max\_size=MAX\_VOCAB\_SIZE,

min\_freq=1)

# 保存词表

pkl.dump(word\_to\_id, open(vocab\_dir, 'wb'))

#读取预训练词向量并更新词表向量

initial\_embedding(pretrain\_dir,word\_to\_id,emb\_dim)

【案例代码】train\_eval.py

import numpy as np

import torch

import torch.nn as nn

import torch.nn.functional as F

from sklearn import metrics#模型评估

import time

from utils import get\_time\_dif

from tensorboardX import SummaryWriter#记录训练过程

# 权重初始化，默认xavier

def init\_network(model, method='xavier', exclude='embedding', seed=123):

#遍历模型中的每一个参数

for name, w in model.named\_parameters():

#排除embedding参数，这个加载预训练词向量

if exclude not in name:

#如果是w参数

if 'weight' in name:

if method == 'xavier':

nn.init.xavier\_normal\_(w)

elif method == 'kaiming':

nn.init.kaiming\_normal\_(w)

else:

nn.init.normal\_(w)

#如果是b参数

elif 'bias' in name:

nn.init.constant\_(w, 0)

else:

pass

def train(config, model, train\_iter, dev\_iter, test\_iter):

start\_time = time.time()

#指定是模型训练模式

model.train()

#实例化优化器

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

# 学习率指数衰减，每次epoch：学习率 = gamma \* 学习率

scheduler = torch.optim.lr\_scheduler.ExponentialLR(optimizer, gamma=0.9)

total\_batch = 0 # 记录进行到多少batch

dev\_best\_loss = float('inf')

last\_improve = 0 # 记录上次验证集loss下降的batch数

flag = False # 记录是否很久没有效果提升

writer = SummaryWriter(log\_dir=config.log\_path + '/' + time.strftime('%m-%d\_%H.%M', time.localtime()))

#遍历num\_epochs次

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

print('Epoch [{}/{}]'.format(epoch + 1, config.num\_epochs))

scheduler.step() # 学习率衰减

#遍历训练集

for i, (trains, labels) in enumerate(train\_iter):

#模型计算输出值

outputs = model(trains)

#梯度清零

model.zero\_grad()

#计算损失

loss = F.cross\_entropy(outputs, labels)

#反相传播计算梯度

loss.backward()

#优化器修改参数

optimizer.step()

if total\_batch % 100 == 0:

# 每多少轮输出在训练集和验证集上的效果

true = labels.data.cpu()

#预测

predic = torch.max(outputs.data, 1)[1].cpu()

#计算准确率

train\_acc = metrics.accuracy\_score(true, predic)

#计算验证集的准确率和损失

dev\_acc, dev\_loss = evaluate(config, model, dev\_iter)

if dev\_loss < dev\_best\_loss:#如果本轮loss小于历史最小loss

dev\_best\_loss = dev\_loss

#保存模型

torch.save(model.state\_dict(), config.save\_path)

improve = '\*'

last\_improve = total\_batch

else:

improve = ''

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

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}'

print(msg.format(total\_batch,

loss.item(),

train\_acc,

dev\_loss,

dev\_acc,

time\_dif,

improve))

#将训练信息写入日志

writer.add\_scalar("loss/train", loss.item(), total\_batch)

writer.add\_scalar("loss/dev", dev\_loss, total\_batch)

writer.add\_scalar("acc/train", train\_acc, total\_batch)

writer.add\_scalar("acc/dev", dev\_acc, total\_batch)

#重新指定为训练模式 因为在上面评估的时候修改成了验证模式

model.train()

total\_batch += 1

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

# 验证集loss超过1000batch没下降，结束训练

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

flag = True

break

if flag:

break

writer.close()

#最后在测试集上评估

test(config, model, test\_iter)

def test(config, model, test\_iter):

# 加载模型

model.load\_state\_dict(torch.load(config.save\_path))

# 指定模型为验证模式

model.eval()

start\_time = time.time()

#测试集评估

test\_acc, test\_loss, test\_report, test\_confusion = evaluate(config, model, test\_iter, test=True)

msg = 'Test Loss: {0:>5.2}, Test Acc: {1:>6.2%}'

print(msg.format(test\_loss, test\_acc))

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

print(test\_report)

print("Confusion Matrix...")

print(test\_confusion)

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

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

def evaluate(config, model, data\_iter, test=False):

"""

评估模型

:param config:配置

:param model: 模型

:param data\_iter: 数据集迭代器

:param test: 是否测试

:return:

"""

# 指定模型为验证模式

model.eval()

loss\_total = 0

predict\_all = np.array([], dtype=int)

labels\_all = np.array([], dtype=int)

with torch.no\_grad():#不计算梯度

for texts, labels in data\_iter:#遍历数据集

outputs = model(texts)#计算输出值

#计算损失

loss = F.cross\_entropy(outputs, labels)

loss\_total += loss

labels = labels.data.cpu().numpy()

#计算预测值

predic = torch.max(outputs.data, 1)[1].cpu().numpy()

labels\_all = np.append(labels\_all, labels)

predict\_all = np.append(predict\_all, predic)

#计算准确率

acc = metrics.accuracy\_score(labels\_all, predict\_all)

if test:#人如果是test测试

#输出分类报告

report = metrics.classification\_report(labels\_all,

predict\_all,

target\_names=config.class\_list,

digits=4)

#输出混淆矩阵

confusion = metrics.confusion\_matrix(labels\_all, predict\_all)

return acc, loss\_total / len(data\_iter), report, confusion

return acc, loss\_total / len(data\_iter)

【案例代码】models/TextRCNN.py

import torch

import torch.nn as nn

import torch.nn.functional as F

import numpy as np

class Config(object):

"""配置参数"""

def \_\_init\_\_(self, dataset, embedding):

self.model\_name = 'TextRCNN'

# 训练集

self.train\_path = dataset + '/data/train.txt'

# 验证集

self.dev\_path = dataset + '/data/dev.txt'

# 测试集

self.test\_path = dataset + '/data/test.txt'

# 类别名单

self.class\_list = [x.strip() for x in open(

dataset + '/data/class.txt', encoding='utf-8').readlines()]

# 词表

self.vocab\_path = dataset + '/data/vocab.pkl'

# 模型保存路径

self.save\_path = dataset + '/saved\_dict/' + self.model\_name + '.ckpt'

#模型训练日志路径

self.log\_path = dataset + '/log/' + self.model\_name

# 预训练词向量

self.embedding\_pretrained = torch.tensor(

np.load(dataset + '/data/' + embedding)["embeddings"].astype('float32'))\

if embedding != 'random' else None

# 设备

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

# 随机失活

self.dropout = 0.5

# 早停 若超过1000batch效果还没提升，则提前结束训练

self.require\_improvement = 1000

# 类别数

self.num\_classes = len(self.class\_list)

# 词表大小，在运行时赋值

self.n\_vocab = 0

# epoch数

self.num\_epochs = 15

# 批次大小

self.batch\_size = 64

# 每句话处理成的长度(短填长切)

self.pad\_size = 100

# 学习率

self.learning\_rate = 1e-3

# 字向量维度, 若使用了预训练词向量，则维度统一

self.embed = self.embedding\_pretrained.size(1)\

if self.embedding\_pretrained is not None else 300

# lstm隐藏层神经元数

self.hidden\_size = 256

# lstm层数

self.num\_layers = 2

'''Recurrent Convolutional Neural Networks for Text Classification'''

#继承nn.Module

class Model(nn.Module):

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

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

#如果使用预训练词向量

if config.embedding\_pretrained is not None:

#构建Embedding层并从预训练词向量中加载参数

self.embedding = nn.Embedding.from\_pretrained(config.embedding\_pretrained,

freeze=False)

else:

#构建Embedding层，不加载参数，后面会随机初始化

self.embedding = nn.Embedding(config.n\_vocab, #词表大小

config.embed,#词嵌入维度

padding\_idx=config.n\_vocab - 1)

#搭建BiLstm层

self.lstm = nn.LSTM(config.embed, #输入维度

config.hidden\_size, #隐藏层维度

config.num\_layers,#层数

bidirectional=True, #是否双向

batch\_first=True,

dropout=config.dropout)

#最大池化层

self.maxpool = nn.MaxPool1d(config.pad\_size)

#全连接层 因为是双向 所以成2 然后加上embed 即： 前向+后向+embed

self.fc = nn.Linear(config.hidden\_size \* 2 + config.embed,

config.num\_classes)

def forward(self, x):

x, \_ = x

# [batch\_size, seq\_len, embeding]

embed = self.embedding(x)

# [batch\_size, seq\_len, hidden\_size \* 2]

out, \_ = self.lstm(embed)

# [batch\_size, seq\_len, hidden\_size \* 2 + embed\_size]

out = torch.cat((embed, out), 2)

out = F.relu(out)

# [batch\_size, hidden\_size \* 2+ embed\_size , seq\_len]

out = out.permute(0, 2, 1)#调换维度

#[batch\_size, hidden\_size \* 2 + embed\_size]

out = self.maxpool(out).squeeze()#池化

#[batch\_size, num\_class]

out = self.fc(out)#全连接

return out

【案例代码】utils.py

import time

import torch

import numpy as np

from train\_eval import train, init\_network

from importlib import import\_module

import argparse

from utils import build\_dataset, build\_iterator, get\_time\_dif

#定义ArgumentParser参数对象

parser = argparse.ArgumentParser(description='Chinese Text Classification')

#指定是否使用预训练词向量

parser.add\_argument('--embedding', default='pre\_trained', type=str, help='random or pre\_trained')

args = parser.parse\_args()

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

dataset = '95588' # 数据集

# 搜狗新闻:embedding\_SougouNews.npz, 随机初始化:random

embedding = 'embedding\_SougouNews.npz'

if args.embedding == 'random':

embedding = 'random'

#加载TextRCNN类

x = import\_module('models.TextRCNN')

#生成配置对象

config = x.Config(dataset, embedding)

np.random.seed(1)

torch.manual\_seed(1)

torch.cuda.manual\_seed\_all(1)

torch.backends.cudnn.deterministic = True # 保证每次结果一样

start\_time = time.time()

print("Loading data...")

#加载数据集 训练集、验证集、测试集

vocab, train\_data, dev\_data, test\_data = build\_dataset(config)

#构建训练集、验证集、测试集迭代器

train\_iter = build\_iterator(train\_data, config)

dev\_iter = build\_iterator(dev\_data, config)

test\_iter = build\_iterator(test\_data, config)

#获取时间差

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

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

# train

#加载词表

config.n\_vocab = len(vocab)

#利用配置参数实例化模型

model = x.Model(config).to(config.device)

#打印模型参数名

print(model.parameters)

#调用训练函数完成训练

train(config, model, train\_iter, dev\_iter, test\_iter)