案例五：医疗命名实体识别

1.案例目的

• 掌握序列标注任务的文本预处理方法；

• 了解命名实体识别中的IOB标注；

• 掌握BiLstm+Crf模型的搭建方法；

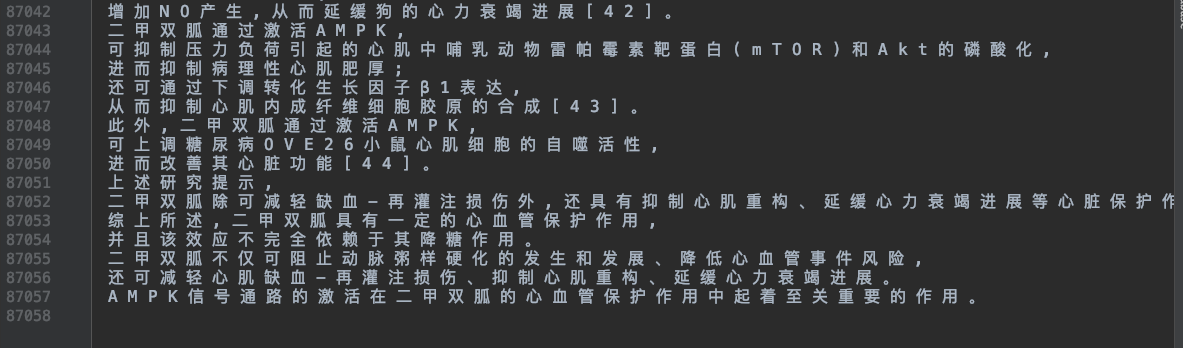
• 掌握BiLstm+Crf模型的训练和评估；

• 使用训练的模型进行命名实体识别预测；

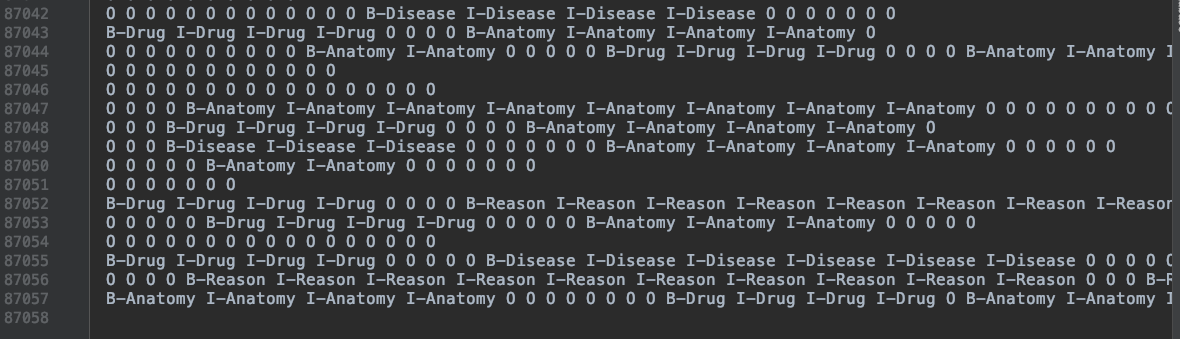
2.案例内容

本次案例旨在通过糖尿病相关的教科书、研究论文来做糖尿病文献挖掘并构建糖尿病知识图谱，我们需要使用已经标注的医学语料，同构搭建BiLstm+Crf进行训练，最终实现使用算法来提取医疗领域的各种实体，从而为进一步提取实体关系构建知识图谱打好基础。

本案例提供的数据已经经过IOB标注处理，并且文本和标注是分为两个文件存放，其中,文本数据已经做了单字空格间隔处理如下图所示:



标注文件如下图所示:



实体类别共十五类。

类别名称和定义

疾病相关：

1、疾病名称 (Disease)，如I型糖尿病。

2、病因(Reason)，疾病的成因、危险因素及机制。比如“糖尿病是由于胰岛素抵抗导致”，胰岛素抵抗是属于病因。

3、临床表现 (Symptom)，包括症状、体征，病人直接表现出来的和需要医生进行查体得出来的判断。如"头晕" "便血" 等。

4、检查方法(Test)，包括实验室检查方法，影像学检查方法，辅助试验，对于疾病有诊断及鉴别意义的项目等，如甘油三酯。

5、检查指标值(Test\_Value)，指标的具体数值，阴性阳性，有无，增减，高低等，如”>11.3 mmol/L”。

治疗相关：

6、药品名称(Drug)，包括常规用药及化疗用药，比如胰岛素。

7、用药频率(Frequency)，包括用药的频率和症状的频率，比如一天两次。

8、用药剂量（Amount），比如500mg/d。

9、用药方法（Method）：比如早晚，餐前餐后，口服，静脉注射，吸入等。

10、非药治疗(Treatment)，在医院环境下进行的非药物性治疗，包括放疗，中医治疗方法等，比如推拿、按摩、针灸、理疗，不包括饮食、运动、营养等。

11、手术（Operation），包括手术名称，如代谢手术等。

12、不良反应（SideEff），用药后的不良反应。

常规实体：

13、部位（Anatomy），包括解剖部位和生物组织，比如人体各个部位和器官，胰岛细胞。

14、程度（level），包括病情严重程度，治疗后缓解程度等。

15、持续时间(Duration)，包括症状持续时间，用药持续时间，如“头晕一周”的“一周”。

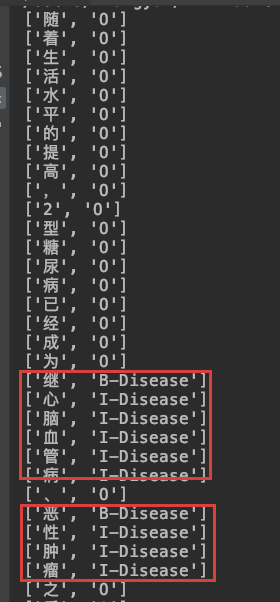
我们以第87050行数据为例:

文本:进 而 改 善 其 心 脏 功 能 [ 4 4 ] 。

标签: O O O O O B-Anatomy I-Anatomy O O O O O O O

根据标签我们可以看到“心脏”是一个“部位（Anatomy）”类型的实体。

在本案例中,模型经过训练后,我们使用模型来提取文本“随着生活水平的提高，2型糖尿病已经成为继心脑血管病、恶性肿瘤之后的又一常见疾病”中的实体，经过模型预测，结果如下:



3.案例知识点

• IOB标注；

• 序列标注文本预处理方式；

• 序列标注训练数据构造；

• BiLstm+CRF模型搭建；

• CRF损失计算方法；

• 使用pytorch进行BiLstm+CRF模型训练；

• 使用pytorch进行命名实体识别；

4.案例时长

共4学时，学时分配如下：

• 数据预处理（1学时）；

• BiLstm+CRF模型搭建（2学时）。

• 模型训练和预测（1学时）

5.案例实验环境

• Windows 10 操作系统；

• Pycharm

• Python3.6

• pytorch 1.6.0

6.案例分析

1）创建项目并拷贝数据文件

2）编写加载数据相关函数

3）实现数据批次迭代器

4）实现BiLstm+CRF完整模型

4.1）实现BiLstm模型

4.2）实现CRF层

4.3）组装BiLstm和CRF

5）实现模型训练和评估

6）实现命名实体识别预测

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“Medical-NER”。

2）创建“dataset”目录

在Medical-NER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。

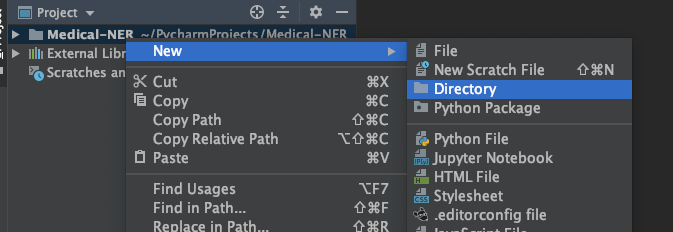


图4 创建目录菜单

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“dataset”；

4）点击“OK”按钮。

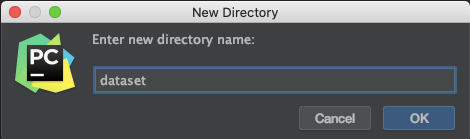


图5 创建目录向导

3）创建其他目录

    按照同样的方法创建目录“checkpoints”、“result”、“scripts”，最终结果如图6所示：

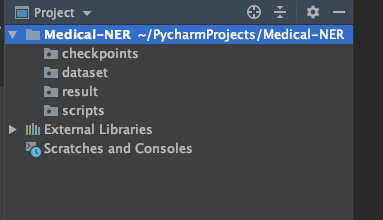


图6 创建目录结果

4）复制文件

将案例提供的数据文件复制到“dataset”目录下，结构如图7所示：

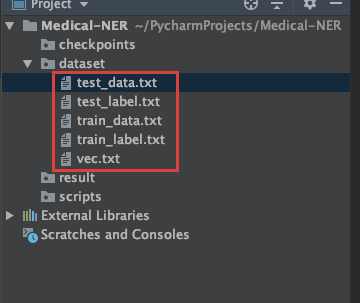


图7 复制数据文件结果

其中各个文件的作用为：

• test\_data.txt：测试集文本数据

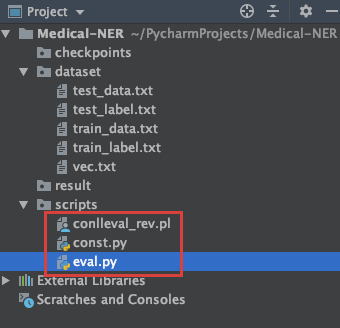
• test\_label.txt：测试集标签数据

• train\_data.txt：训练集文本数据

• train\_label.txt：测试集标签数据

• vec.txt：预训练字向量文件

将案例提供的脚本文件复制到“scripts”目录下，结构如下图所示：



其中各文件作用为：

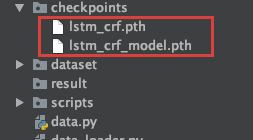
• conlleval\_rev.pl：该文件为perl语言脚本文件，其功能是对模型预测的实体结果与真实结果进行统计，并得到分类报告。

• const.py： 该文件中保存的是本案例中涉及的所有实体标注标签类别字典，由于内容较多，所以预先给出，其内容如下：



• eval.py：该文件为调用conlleval\_rev.pl脚本的代码。

将案例提供的预训练模型文件复制到“checkpoints”目录下，结构如下图所示：



7.2编写加载数据相关函数

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。为了方便加载数据、词典以及预训练自向量，我们需要实现以下几个函数：

• read\_corpus：该函数负责加载数据并对数据进行清洗和预处理；

• read\_dictionary：该函数负责读取词典文件并返回字典word2id；

• get\_vector：该函数负责读取停用词文件并返回词向量字典；

首先，在Medical-NER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：

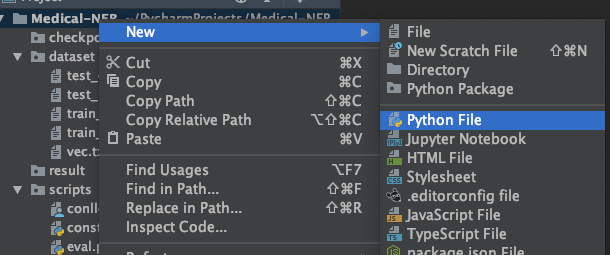


图8 新建py文件菜单

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“data”；

4）点击“OK”按钮。

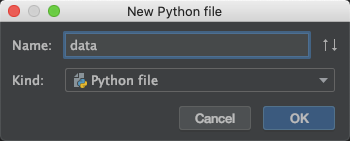


图9 新建py文件向导

在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:

• pickle：用来读取二进制字典文件，这里我们使用该包读取词典文件;

• os：用来拼接路径；

• scripts.const：导入我们需要使用的tag2label标签字典

【代码1】data.py

#导入使用的包

import pickle, os

from scripts.const import \*

2）编写read\_corpus函数代码

该函数参数有三个：

• corpus\_path：文本数据文件路径；

• label\_path：对应标签文件路径；

• is\_train：是训练还是测试，训练阶段处理方式跟测试阶段处理方式不同；

在read\_corpus方法中，我们主要进行了以下工作：

• 读取文本数据文件和对应的标签文件；

• 读取或者创建词典；

• 这个环节中，我们需要用到一个函数read\_dictionary，该函数负责读取词典文件，将在下一步中创建

• 遍历文本和标签的每一行，对数据进行简单清洗，并使用词典和标签词典分别将字和标签转换成对应的id；

• 如果是训练阶段，则在构建的词典加入两个特殊字符（<UNK>：未登录词，<PAD>：填充词），然后保存到文件；

下面，我们首先实现文件读取和词典加载/创建部分代码，在data.py文件中输入如下代码：

【代码2】data.py

#加载数据并进行预处理

def read\_corpus(corpus\_path, label\_path, is\_train=True):

sents = []#保存所有句子

labels = []#保存所有标签

data = []#保存所有数据

with open(corpus\_path, encoding='utf-8') as fr:#打开文本数据文件

lines\_co = fr.readlines()#读取所有行

with open(label\_path) as fl:#打开标签文件

lines\_lb = fl.readlines()#读取所有行

if not is\_train:#如果不是训练，就加载已经创建好的词典文件

word2id = read\_dictionary('./dataset/vocab.pkl')

else:#如果是训练，则需要重新创建词典文件

word2id = {}

读取了数据和标签之后，我们需要使用循环同时遍历两个文件（两个文件内容是对应的），并对读取的每一行文本数据和标签进行如下处理：

• 取出当前行文本两端空格后判断当前行文本是否为空，为空则不处理，继续下一行

• 判断每一个字是不是数字，是数字就转换成特殊符号<NUM>（所有数字看成同一个字符，可以提高模型在数字上的泛化能力）

• 判断是训练还是测试：

• 是训练，则判断在不在词典内，不在就加入词典，这个过程就是在创建词典；

• 是测试，则判断字在不在词典内，不在则转换成特殊符号<UNK>（表示该字是未知字，解决未登录词问题）

• 将字转换成对一个的id并加入列表

• 将每一个标签转换成对应的id加入列表

• 将句子和标签序列加入总列表中

【代码3】data.py

for line\_co, line\_lb in zip(lines\_co, lines\_lb):

sent\_,tag\_ = line\_co.strip().split(),line\_lb.strip().split()#去空格

if len(sent\_)==0:#句子长度为0 就不处理

continue

data.append((sent\_, tag\_))#原数据放入data

sentence\_id,label\_ = [],[]#保存词id构成的序列和标签id构成的序列

for word in sent\_:

if word.isdigit():#判断是数字就转换成<NUM>

word = '<NUM>'

if is\_train:

if word not in word2id:

word2id[word] = len(word2id)+1#加入词典

else:

if word not in word2id:#不再词典内就是未知词

word = '<UNK>'

sentence\_id.append(word2id[word])#转换成id加入列表

for tag in tag\_:

label = tag2label[tag]

label\_.append(label)

sents.append(sentence\_id)#保存到所有句子中

labels.append(label\_)#保存到所有标签序列中

最后，我们需要将前面创建的词典保存到文件中以便测试的时候读取使用，具体步骤如下：

• 判断是否训练，训练才做以下操作；

• 向词典中添加特殊符号<UNK>和<PAD>；

• 使用pickle将词典保存为二进制文件

• 返回构建好的sents，labels和data

【代码4】data.py

if is\_train:#判断是否是训练，是训练才保存词典

word2id['<UNK>'] = len(word2id)+1#词典中加入<UNK>

word2id['<PAD>'] = PAD#词典中加入<PAD>

print('vocabulary length:', len(word2id))

with open('./dataset/vocab.pkl', 'wb') as fw:

pickle.dump(word2id, fw)#将词典保存

return sents, labels, data#返回导入结果

3）编写读取词典的函数read\_dictionary。

在上一步实现的函数read\_corpus中我们需要调用一个用来读取词典文件的函数read\_dictionary，read\_dictionary函数的功能很简单，就是根据给定的词典文件路径将词典加载并返回。其参数为：

• vocab\_path：词典文件路径

我们继续在data.py文件中编写该函数，代码如下：

【代码5】data.py

def read\_dictionary(vocab\_path):

vocab\_path = os.path.join(vocab\_path)#拼接路径

with open(vocab\_path, 'rb') as fr:

word2id = pickle.load(fr)#加载词典

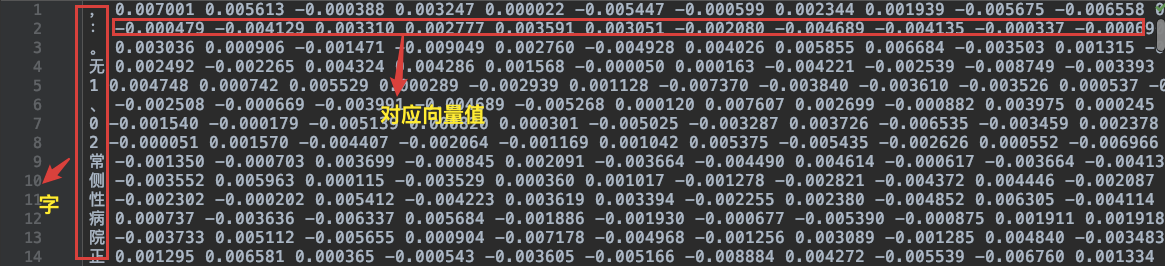
return word2id#返回词典

4）编写加载预训练词向量的函数get\_vector

本案例中，我们将采用预训练的字向量来对文本进行表示，因此在模型初始化的时候需要将预训练字向量加载，这里我们先将函数边写出来。该函数的参数为：

• path：预训练字向量文件路径

我们先双击打开7.1节复制的dataset目录下的vec.txt文件，观察文件内容如下图所示：



其中：每一行的第一个字符为单个汉字符号，后面的数值即为该字对应的字向量，长度为100，我们需要编写函数get\_vector实现将该文件读取并解析成如下的字典形式：{字：向量}，该函数主要步骤如下：

• 打开文件，遍历每一行

• 去掉两端空格后使用空格拆分

• 第一个字符是字，后面的都是对应的向量值

• 将向量转换成float类型

• 判断向量长度==100就将字和向量加入字典

【代码6】data.py

def get\_vector(path):

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

vectors={}#空字典 用来保存读取的词向量

for line in f.readlines():

line=line.strip().split(' ')#空格分隔

word,vector = line[0],line[1:]#第一个是字，后面的是向量的值

vector=[float(i) for i in vector]#将数值字符串转换成小数

if len(vector)==100:#长度为100才对

vectors[word]=vector#放入字典

return vectors#返回

• 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；

7.3实现数据批次迭代器

深度学习模型在训练时，由于内存的限制，我们需要将数据分批次送入模型进行训练，因此我们需要编写一个类DataLoader来对数据进行批次迭代管理。在该类中，我们主要完成以下工作：

• 数据增强，本案例中，我们将训练数据中前后两个句子拼接得到长句子来增加训练数据；

• 数据清除，由于训练数据中存在部分实体标注全是“O”的句子，这种句子是纯粹的负样本，对实体的识别没有作用，而且这些句子的存在还会导致训练的速度变慢，因此我们将这样的句子去掉；

• 计算批次数量，批次数量=数据数量/批次大小，但需要注意有可能不能整除，还需要做一下处理；

• 对数据进行随机打乱或者排序

• 实现\_\_next\_\_方法生成批次数据并对同一批数据进行填充

在项目Medical-NER中新建Python File文件“data\_loader”。

1）导入需要使用的包。

【代码9】data\_loader.py

import numpy as np

import torch

from scripts.const import \*

2）编写初始化方法\_\_init\_\_

我们首先实现DataLoader类的\_\_init\_\_初始化方法，通过实现该方法我们先理解一下整个数据处理的过程，然后再编写实现各个环节中用到的函数。首先看一下该方法的参数：

• sents：整个数据集的所有句子（已经id化）

• labels：整个数据集句子对应的标签

• cuda：是否使用gpu

• batch\_size：批次大小

• shuffle：指定随机打乱数据还是按长度排序数据

• evaluation：指定是否是测试状态

在该方法中，我们队数据进行如下处理：

• 对类属性进行初始化

• 判断不是测试就进行数据增强

• 将sents和labels转换成numpy的ndarray数组

• 判断不是测试就进行数据清理（去掉都是O的句子）

• 进行批次计算，批次数量=数据量/批次大小

• 能整除，批次数量=int（数据量/批次大小）

• 不能整除，则需要在得到商之后再加1，即批次数量=int（数据量/批次大小）+1

• 对数据进行随机打乱顺序或者按长度排序

我们再data\_loader.py文件中编写如下代码：

【代码9】data\_loader.py

class DataLoader(object):

def \_\_init\_\_(self, sents,labels, cuda=True,batch\_size=64, shuffle=False, evaluation=False):

self.cuda = cuda

self.sents=sents

self.labels=labels

self.evaluation = evaluation

self.\_batch\_size = batch\_size

if not evaluation:

self.\_augment()#实现数据增强

self.sents,self.labels = np.asarray(self.sents),np.asarray(self.labels)

if not evaluation:

self.\_clean()#实现数据清理

self.sents\_size = len(self.sents)

self.\_step = 0#用来记录迭代时的批次下标

self.num\_batchs = self.sents\_size // batch\_size#计算批次数量

self.divided=True#记录是否整除

if self.sents\_size%batch\_size!=0:#不能整除

self.divided=False

self.num\_batchs+=1#批次数量+1

if shuffle:

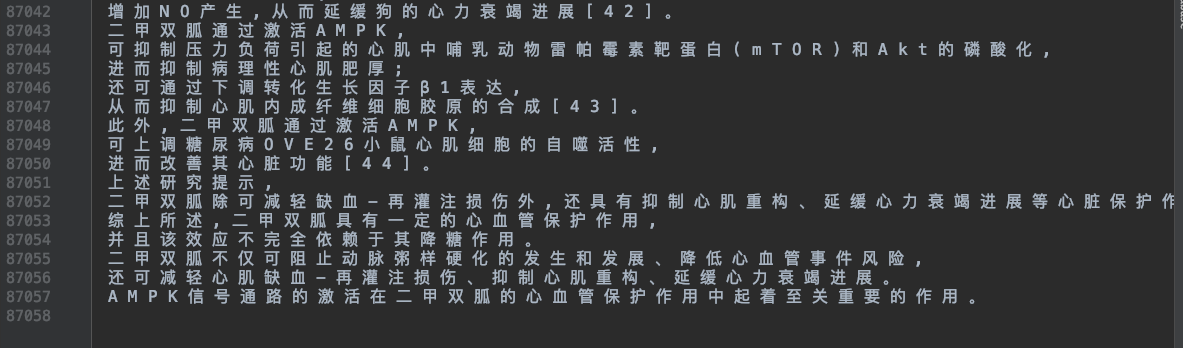
self.\_shuffle()#打乱数据

elif not evaluation:

self.\_sorted()#排序数据\*

3）编写实现数据增强的方法\_augment

我们首先观察一下数据文件：



通过观察可以发现，文本中的每一行其实本身是一篇文章中的句子，前后两行之间是可以连贯的句子，因此，我们可以来考虑将前后两个句子拼接成一个长的句子，这样可以增加数据量和数据多样性。\_augment方法的实现很简单，只需要将sents（句子）和labels（对应的标签序列）分别进行前后两个拼接即可。代码如下：

【代码10】data\_loader.py

def \_augment(self):

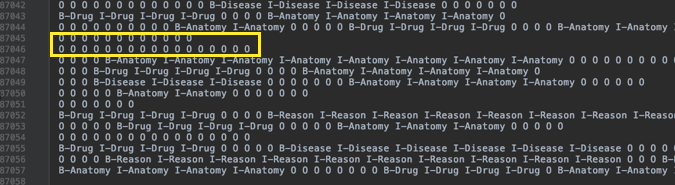
#前后两句拼接成一个句子

self.sents=self.sents+[self.sents[i]+self.sents[i+1] for i in range(len(self.sents)-1)]#句子拼接后加入句子列表中

self.labels = self.labels + [self.labels[i] + self.labels[i + 1] for i in range(len(self.labels) - 1)]#标签拼接

4）编写实现数据清理的方法\_clean

我们再来观察一下标签文件：



我们可以看出，其中有的句子标注的标签都是O，意思就是当前句子中没有一个实体，因而这样的句子样本是纯粹的负样本；同时在命名实体识别训练数据中，负样本本身就已经很多，因此我们可以去掉这样的句子，提高训练速度的同时还可以代码

【代码11】data\_loader.py

def \_clean(self):

#删除标签全部是O的句子,就是没有实体的句子

sums=[len(set(item))>1 for item in self.labels]#计算句子中非O的个数

indices=np.where(sums)[0]#筛选出全是O的句子

self.sents=self.sents[indices]#布尔索引实现文本筛选

self.labels = self.labels[indices]#对应的标签也要去掉

5）编写实现数据随机打乱的方法\_shuffle

该方法的目的是将训练数据随机打乱，从而增强模型训练时的稳定性，其思路为：

• 使用np.random.permutation生成随机打乱后的数据下标

• 使用乱序的下标对sents和labels进行重新索引

代码如下:

【代码11】data\_loader.py

def \_shuffle(self):

indices=np.random.permutation(self.sents\_size)#获取乱序下标

self.sents = self.sents[indices]#文本数据重新索引

self.labels = self.labels[indices]#标签数据重新索引

6）编写实现数据排序的方法\_sorted

由于训练文本的长短不一，同一批次的文本我们需要填充到该批次最长文本的长度，加入一个批次的文本长度为【3，6，50，4】，那么我们需要在短的文本后面添加<PAD>将其填充值最长的文本长度，填充后长度为【50，50，50，50】，然后送入模型训练。因此，如果一个批次中有一个文本很长很长，就讲导致该批次文本填充大量的<PAD>，最终导致训练速度变慢。

我们可以考虑将文本按照长度排序，这样长度相似的文本就会被分到一个批次中，填充量就少了很多。我们可以对比一下：

• 不排序：【3，6，50，4】 填充后【50，50，50，50】

• 排序：【3，4，6，8】 填充后【8，8，8，8】

可以发现，排序后在分批次填充，填充量小了很多，这样可以加快训练的速度。

【代码11】data\_loader.py

def \_sorted(self):

lens=np.array([len(sent) for sent in self.sents])#计算句子长度

indices=np.argsort(lens)#排序

self.sents = self.sents[indices]#重新索引

self.labels = self.labels[indices]#重新索引

7）编写实现数据填充的方法\_pad\_to\_longest

该函数接收一个批次的数据，然后将该批数据序列都填充到该批次的最大长度：

• 填充前：【3，6，50，4】 填充后：【50，50，50，50】

其流程为：

• 计算每个序列长度

• 获取最大长度

• 在每个序列的后面填充<PAD>，将其填充到最大长度

• 构建成torch的tensor

【代码12】data\_loader.py

def \_pad\_to\_longest(self,insts):

seq\_len\_list = [len(inst) for inst in insts]#计算长度

max\_len = max(len(inst) for inst in insts)#获取最大长度

inst\_data = np.array(

[inst + [PAD] \* (max\_len - len(inst)) for inst in insts])#填充

if self.evaluation:#测试情况下

with torch.no\_grad():#不求导

inst\_data\_tensor = torch.from\_numpy(inst\_data)#构建成tensor

else:

inst\_data\_tensor = torch.from\_numpy(inst\_data)

if self.cuda:#如果开启cuda

inst\_data\_tensor = inst\_data\_tensor.cuda()

return inst\_data\_tensor, seq\_len\_list

7）编写实现数据批次迭代的方法\_\_next\_\_和\_\_iter\_\_

\_\_next\_\_函数实现的功能是将数据按照\_batch\_size分成批次，每迭代一次返回一个批次的数据，通过self.\_step和self.num\_batchs来控制迭代继续还是结束。逻辑如下：

• 如果self.\_step == self.num\_batchs，即迭代步数等于批次数，说明已经迭代完一轮，那么终止迭代

• 如果self.\_step == self.num\_batchs-1 and self.divided==False，即迭代步数等于批次数量-1，意味着是最后一个批次，最后一个批次要看数据总数能否被批次大小整除，self.divided==False表示不能整除，那么最后一个批次就没有\_batch\_size个数据，就返回余数个。反之，能整除，还是返回\_batch\_size个。

• 然后，步数+1，并对该批次的文本序列和标签序列进行填充；

• 最后，返回【文本，标签，真实长度】

代码如下：

【代码11】data\_loader.py

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

return self

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

if self.\_step == self.num\_batchs:#步数等于批次数

self.\_step = 0

raise StopIteration()#终止迭代

\_start = self.\_step \* self.\_batch\_size

if (self.\_step == self.num\_batchs-1 and self.divided==False):#最后一批且不能整除

\_sp=self.sents\_size%self.\_batch\_size#返回余数个

else:

\_sp=self.\_batch\_size

self.\_step += 1

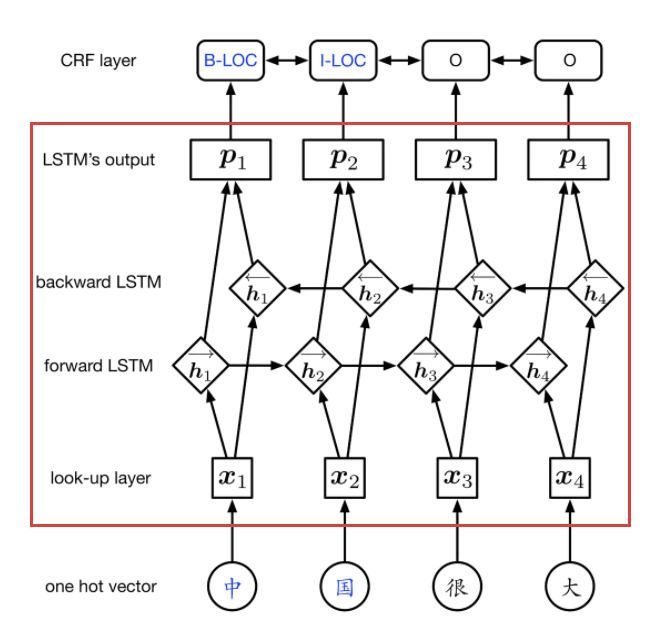
word, seq\_len\_list = self.\_pad\_to\_longest(self.sents[\_start:\_start + \_sp])#填充文本

label, \_ = self.\_pad\_to\_longest(self.labels[\_start:\_start + \_sp])#填充标签

return word, label, seq\_len\_list

7.4实现BiLstm+CRF完整模型

本次案例我们使用的模型为BiLstm+CRF模型，期模型架构图如下图所示：



该模型架构中主要包含两个部分：

• BiLstm层，图中红框的部分就是BiLstm层，该层的功能是将每个token的输入转换为融合了上下文的token输出；

• CRF层，CRF层可以为最后预测的标签添加一些约束来保证预测的标签是合法的。在训练数据训练过程中，这些约束可以通过CRF层自动学习到的。

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Medical-NER项目中新建Python File文件“model”。

1）导入需要使用的包。

【代码12】model.py

import torch

import torch.nn as nn

from torch.nn import init

import numpy as np

from scripts.const import \*

2） 编写类BiLSTM的初始化方法\_\_init\_\_

初始化方法接收的参数有：

• word\_ebd\_dim：词向量嵌入维度

• lstm\_hsz：lstm的神经元数量

• lstm\_layers：lstm层数

• dropout：随机失活比例

• batch\_size：批次大小

• vec：预训练字向量字典

• word2id：词典

• device：设备 gpu/cpu

初始化方法中我们主要的工作是实现结构中的两个层：

• 实现Embedding层

• 判断是否给定字向量，给定则加载预训练字向量创建嵌入层

• 没有给定，则随机初始化嵌入层

• 实现BiLstm层

【代码13】model.py

class BiLSTM(nn.Module):

def \_\_init\_\_(self, word\_ebd\_dim, lstm\_hsz, lstm\_layers, dropout, batch\_size,vec,word2id,device):

super().\_\_init\_\_()

self.lstm\_layers = lstm\_layers

self.lstm\_hsz = lstm\_hsz

self.batch\_size = batch\_size

self.word2id=word2id

self.word\_size=len(word2id)

self.word\_ebd\_dim=word\_ebd\_dim

self.device=device

self.scope=1

self.vec=vec

if vec:#如果预训练字向量存在

vecs =self.\_get\_vec()#获取字向量

self.word\_ebd = nn.Embedding.from\_pretrained(vecs)#使用预训练的字向量实例化嵌入层

else:

self.word\_ebd = nn.Embedding(self.word\_size, self.word\_ebd\_dim)#随机初始化嵌入层

self.word\_ebd.weight.data.uniform\_(self.scope, self.scope)

self.lstm = nn.LSTM(word\_ebd\_dim,

hidden\_size=lstm\_hsz // 2,

num\_layers=lstm\_layers,

batch\_first=True,

dropout=dropout,

bidirectional=True)#实现BiLstm层

3）编写预训练字向量初始化方法\_get\_vec。

该方法中，我们主要目的是使用预训练的字向量来对我们词表中的词进行向量化嵌入，工作流程为：

• 首先随机初始化词表向量权重weight

• 然后，遍历给定的字向量字典vec，

• 如果字k在此表中出现，就是用k对应的预训练向量v对weight中的对应向量进行更新

• 返回更新后的向量权重weight

【代码14】model.py

def \_get\_vec(self):

weight = torch.Tensor(self.word\_size, self.word\_ebd\_dim).uniform\_(-self.scope,self.scope)

for k,v in self.vec.items():

if k in self.word2id.keys():

index = self.word2id[k]#获取词对应的id

weight[index, :] = torch.from\_numpy(np.array(v))#将该词对应的嵌入向量修改为预训练向量

return weight

4）编写初始化lstm初始状态的方法init\_hidden。

该方法的作用是生成lstm中的初始状态向量，后续每次lstm计算时都采用该初始状态向量，这里我们使用正态分布初始化方式进行初始化。状态向量有两个：

• h：shape为【self.lstm\_layers \* 2，self.batch\_size，self.lstm\_hsz // 2】

• c：shape也为【self.lstm\_layers \* 2，self.batch\_size，self.lstm\_hsz // 2】

乘2是因为我们是双向lstm

【代码15】model.py

def init\_hidden(self,batch\_size):

return (torch.zeros(self.lstm\_layers \* 2, batch\_size, self.lstm\_hsz // 2).to(self.device),

torch.zeros(self.lstm\_layers \* 2, batch\_size, self.lstm\_hsz // 2).to(self.device))

5）重写父类Module的forward方法。

该方法为重写的父类方法，在该方法中实现bilstm的计算过程，过程为：

• 嵌入层嵌入：接收words【batch\_size，length】输出encode【batch\_size，length，embed\_size】

• 每次根据传进来的数据批次大小来初始化hidden状态

• lstm计算：接收encode【batch\_size，length，embed\_size】，输出output【batch\_size，length，self.lstm\_hsz】

• 返回output

【代码15】model.py

def forward(self, words):

encode = self.word\_ebd(words)#嵌入计算

#初始化状态

hidden=self.init\_hidden(words.shape[0])

output, hidden = self.lstm(encode,hidden)#sltm计算

return output, hidden

至此，我们完成了BiLSTM类的实现。下一步，我们来实现CRF层。

CRF层主要目的是使用crf马尔科夫线性链来让模型学习到约束的规则，这些约束可以是：

• I：句子中第一个词总是以标签“B-“ 或 “O”开始，而不是“I-”

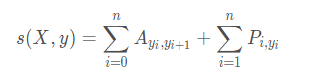
• II：标签“B-label1 I-label2 I-label3 I-…”,label1, label2, label3应该属于同一类实体。例如，“B-Person I-Person” 是合法的序列, 但是“B-Person I-Organization” 是非法标签序列.

• III：标签序列“O I-label” is 非法的.实体标签的首个标签应该是 “B-“ ，而非 “I-“, 换句话说,有效的标签序列应该是“O B-label”。

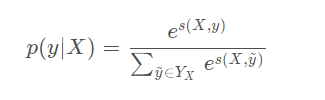
有了这些约束，标签序列预测中非法序列出现的概率将会大大降低。这些约束的实现是通过CRF损失函数来实现的。

假设我们的标签一共有tag\_size个，那么BiLSTM的输出维度就是tag\_size，表示的是每个词wi​映射到tag的发射概率值(feats)，设BiLSTM的输出矩阵为P,其中Pi,j代表词wi映射到tag j的非归一化概率。对于CRF来说，我们假定存在一个转移矩阵A，则Aij代表代表代表tagi转移到转移到转移到tagj的转移概率；

对于输入序列X对应的输出tagtagtag序列y，定义分数为：



利用softmax函数，我们为每一个正确的tag序列y定义一个概率值

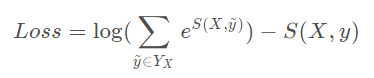


在训练中，我们的目标就是最大化概率p(y│X) ，怎么最大化呢，用对数似然（因为p(y│X)中存在指数和除法，对数似然可以化简这些运算）

对数似然形式如下：



所以我们将损失函数Loss定义为-log(p(y|X))，就可以利用梯度下降法来进行网络的学习了。



这里有两个重要矩阵：

• feats: 发射矩阵(emit score)是sentence 在embedding后,再经过LSTM后得到的矩阵(也就是LSTM的输出), 假设维度为11 \* 5 (11为sentence 的length，5是标签数）。这个矩阵表示经过LSTM后sentence的每个word对应的每个labels的得分)。 表示发射概率。

• self.transitions:转移矩阵，假设维度为55，transitions[i][j]表示label j转移到label i的概率。transtion[i]维度为55,表示每个label转移到label i的概率。 表示概率转移矩阵

1） 编写类CRF的初始化方法\_\_init\_\_

初始化方法接收的参数有：

• label\_size：tag标签的个数

• device：设备gpu/cpu

该方法中主要工作是初始化转移矩阵transitions。其中要注意的是：

• 起始行的权重参数设为-10000，表示tag转移到START的概率为0，因为START只能在开头出现，在后面是不可能出现的；

• 终止列权重参数设为-10000，表示STOP转移到其他tag的概率为0

【代码16】model.py

class CRF(nn.Module):

def \_\_init\_\_(self, label\_size, device):

super().\_\_init\_\_()

self.label\_size = label\_size

self.device=device

self.transitions = nn.Parameter(

torch.randn(label\_size, label\_size)) # 定义转移矩阵

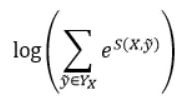
init.xavier\_uniform\_(self.transitions) # 初始化

self.transitions.data[START, :].fill\_(-10000.) # 起始行全部设为-1000

self.transitions.data[:, STOP].fill\_(-10000.) # 终止列全部设为-1000

2） 编写实现log\_sum计算的方法\_log\_sum\_exp

该方法的功能是在前向概率计算过程中负责log\_sum\_exp计算，公式如下：



接收的参数有：

• input：输入的矩阵，二维，【batch\_size，num\_tags】

• keepdim：求和时是否保持维度

其计算过程如下：

• 计算向量最大值max\_scores

• 向量减去最大值input - max\_scores，目的是防止指数运算上溢

• 按照公式计算结果并加上max\_scores后返回

代码如下：

【代码17】model.py

def \_log\_sum\_exp(self,input, keepdim=False):

assert input.dim() == 2

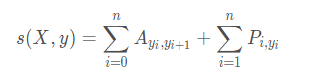
max\_scores, \_ = input.max(dim=-1, keepdim=True)# max\_score维度是１

output = input - max\_scores#减去最大值，谜底是防止指数运算上溢

return max\_scores + torch.log(torch.sum(torch.exp(output), dim=-1, keepdim=keepdim))

3） 编写实现计算当前路径分值的方法\_score\_sentence

该方法实现下面的公式计算：



该方法的功能是根据lstm的输出input（P），crf转移矩阵transitions（A）和真实tag序列来计算当前真实tag的路径分值，其参数为

• input，lstm的输出，shape为【batch\_size，length，tag\_nums】。

• tags，真实标签序列，shape为【batch\_size，length】

其计算过程如下：

• 初始化score，值为0

• 生成并拼接起始符START的tag

• 遍历每一个token的tag，不断累加每个token的转移概率和发射概率

• 最后加上终止符的转移概率

代码如下：

【代码19】model.py

def \_score\_sentence(self, feats, tags):

score = torch.zeros(tags.shape[0]).to(self.device)

s\_score = torch.LongTensor([[START]] \* tags.shape[0]).to(self.device)#初始化分值

tags = torch.cat([s\_score, tags], dim=-1)

for i in range(feats.shape[1]):#遍历每一个token

feat=feats[:,i,:]#获取当前token的发射概率

#加上转移概率和发射概率

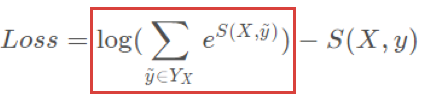
score = score + self.transitions[tags[:,i + 1], tags[:,i]] + feat[range(feat.shape[0]),tags[:,i + 1]]

score = score + self.transitions[STOP, tags[:,-1]]

return score

4） 编写实现计算所有路径分值的方法forward。

该方法实现所有路径分值的计算，即下图中红框部分：



其参数是input，是lstm的输出，shape为【batch\_size，length，tag\_nums】

具体步骤如下：

• 初始化分值矩阵

• 将输入input转置

• 遍历每一个token

• 遍历下一个所有可能的tag，计算转移概率、发射概率并加和进行log\_sum\_exp计算

• 循环结束后，加上终止符的概率分值

代码如下：

【代码20】model.py

def forward(self,input):

# 初始化路径分值

init\_vvars = torch.full((1, self.label\_size), -10000.).to(self.device)

init\_vvars[0][START] = 0

forward\_var\_list = []

forward\_var\_list.append(torch.stack([init\_vvars] \* input.shape[0]).squeeze(1)) # 第一个时刻

for feat\_index in range(input.shape[1]):

gamar\_r\_l = forward\_var\_list[feat\_index].unsqueeze(1) # [128,1,33]

next\_tag\_var = gamar\_r\_l + self.transitions # 128,33,33 加上转移概率

t\_r1\_k = torch.unsqueeze(input[:, feat\_index, :], -1) # 128,33,1 发射概率

forward\_var\_new = next\_tag\_var + t\_r1\_k # 128,33,33 加上发射概率

forward\_var\_list.append(torch.logsumexp(forward\_var\_new, dim=2))#求和

terminal\_var = forward\_var\_list[-1] + self.transitions[STOP].repeat([input.shape[0], 1])

return torch.logsumexp(terminal\_var, dim=1)

5） 编写实现viterbi解码的方法viterbi\_decode。

该方法使用viterbi算法来预测最优路径，即通过biterbi算法找到分值最大的路径，算法原理这里我们不做讲解。方法接收的参数为：

• input：是lstm的输出，shape为【batch\_size，length，tag\_nums】

下面简单说一下计算流程；

• 构建初始分值矩阵

• 将输入input转置

• 遍历每一个token

• 遍历下一个所有可能的tag，计算转移概率，并得到当前时刻每个tag最大的路径分值（跟上一步不同，上一步是前向算法，是求和）

• 得到最后一个时刻的最高分值tag后，开始回溯，不断回溯记录上一步转移过来的tag的id，最终得到最优路径

代码如下：

【代码21】model.py

def \_viterbi\_decode(self, feats):

backpointers = []

#初始化路径分值

init\_vvars = torch.full((1, self.label\_size), -10000.).to(self.device)

init\_vvars[0][START] = 0

forward\_var\_list = []

forward\_var\_list.append(torch.stack([init\_vvars] \* feats.shape[0]))#第一个时刻

for feat\_index in range(feats.shape[1]):

gamar\_r\_l = forward\_var\_list[feat\_index]#[128,1,33]

next\_tag\_var = gamar\_r\_l + self.transitions#128,33,33 加上转移概率

t\_r1\_k = torch.unsqueeze(feats[:,feat\_index,:], -1)#128,33,1 发射概率

forward\_var\_new = next\_tag\_var + t\_r1\_k#128,33,33 加上发射概率

#取出当前时刻的最大分值状态集状态下标

viterbivars\_t, bptrs\_t = torch.max(forward\_var\_new, dim=-1) # 128,33 128,33

forward\_var\_list.append(viterbivars\_t.unsqueeze(1))

backpointers.append(bptrs\_t)

#转移到最后的stop

terminal\_var = forward\_var\_list[-1] + self.transitions[STOP]

# 找到最大路径终止时刻tag下标

path\_score,best\_tag\_id = torch.max(terminal\_var.squeeze(1), dim=-1)

#回溯解码

best\_tag\_id=best\_tag\_id.unsqueeze(1)

best\_path = [best\_tag\_id]

for bptrs\_t in reversed(backpointers):

best\_tag\_id = torch.gather(bptrs\_t,1,best\_tag\_id)#获取上一时刻路径状态

best\_path.append(best\_tag\_id)

best\_path.pop()#START不要

best\_path.reverse()#逆转

return path\_score, torch.cat(best\_path, dim=-1)

至此，CRF层实现完毕，我们主要实现了三个计算部分：

• 真实路径得分计算

• 所有路径得分和的计算—前向算法

• 最优路径预测—viterbi算法

实现了BiLstm和CRF层后，我们需要将二者组装成最终的模型。下面我们定义一个类Model，在该类中，我们封装整个模型。

1）编写类Model的初始化方法\_\_init\_\_。

该方法功能是接收配置参数，并实例化模型中的各个层，我们需要实例化三个层：

• BiLSTM层：将输入的序列【batch\_size，length】映射成【batch\_size，length，lstm\_dim】

• Linear层：该层负责将每个token的输出映射到label的类别数来做分类，【lstm\_dim】【label\_size】

• CRF层：该层负责计算CRF损失

\_\_init\_\_方法接收一个参数字典args，该字典中存储的是模型的配置参数。我们继续在model.py文件中编写如下代码：

【代码24】model.py

class Model(nn.Module):

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

super().\_\_init\_\_()

for k, v in args.\_\_dict\_\_.items():#参数赋值

self.\_\_setattr\_\_(k, v)

self.bilstm = BiLSTM(self.word\_ebd\_dim,

self.lstm\_hsz, self.lstm\_layers,

self.dropout, self.batch\_size,

self.pre\_trained\_vec,self.word2id,self.device

)#bilstm层

self.logistic = nn.Linear(self.lstm\_hsz, self.label\_size)#Linear层

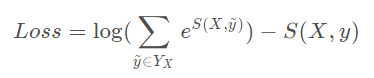
self.crf = CRF(self.label\_size, self.device)#crf层

torch.nn.init.xavier\_normal\_(self.logistic.weight)#初始化linear参数

self.logistic.bias.data.fill\_(0)

2）编写计算crf损失值的方法forward。

该方法最终实现如下公式的计算：



该方法接收两个参数：

• words：输入的文本序列

• labels：真实标签序列

计算流程如下：

• bilstm层计算，得到token向量

• linear层计算，得到label的概率分布，就是发射概率

• crf层计算所有路径分值之和，即公式中的第一项

• crf层计算真实路径分值，即公式中的第二项

• 最后，返回损失值

编写代码如下：

【代码25】model.py

def forward(self, words, labels, hidden=None):

output, \_ = self.bilstm(words)#bilstm计算

output = self.logistic(output)#linear层计算

pre\_score = self.crf.forward(output)#所有路径分值和

label\_score = self.crf.\_score\_sentence(output, labels)#

return (pre\_score - label\_score).mean(), None

3）编写实现预测的方法predict

该方法功能为实现预测过程，接收一个words参数，即输入的文本序列，计算过程如下：

• bilstm层计算

• linear层计算

• crf层viterbi解码预测

编写代码如下：

【代码26】model.py

def predict(self, word):

lstm\_out, \_ = self.bilstm(word)

out = self.logistic(lstm\_out)

return self.crf.viterbi\_decode(out)

至此，我们完成了BiLstm+CRF模型的搭建工作，下面就可以开始编写训练函数进行训练了。

7.5实现模型训练和评估

数据和模型都准备好了之后，我们可以开始来编写执行模型训练的相关代码，我们分以下几个步骤完成：

• 新建文件并导入需要使用的包

• 使用argparse配置参数

• 编写执行训练的函数train

• 编写执行评估的函数evaluate

• 编写主过程代码实现训练和评估

在Medical-NER项目中新建Python File文件“main”。

1）导入需要使用的包。

【代码28】main.py

import argparse

import torch

from model import Model

from data\_loader import DataLoader

from data import read\_corpus, tag2label,read\_dictionary,get\_vector

import os

from eval import conlleval

import time

2）使用argparse配置参数，参数比较多，这里我们不一一介绍，每个参数都有对应的解释，参数配置完成后指定随机数种子和设备。

参数代码比较多，建议直接复制：

编写代码如下：

【代码29】main.py

parser = argparse.ArgumentParser(description='LSTM\_CRF')

parser.add\_argument('--epochs', type=int, default=32,

help='number of epochs for train')

parser.add\_argument('--batch-size', type=int, default=64,

help='batch size for training')

parser.add\_argument('--seed', type=int, default=1111,

help='random seed')

parser.add\_argument('--use-cuda', action='store\_true',default=False,

help='enables cuda')

parser.add\_argument('--lr', type=float, default=0.001,

help='learning rate')

parser.add\_argument('--use-crf', action='store\_true',

help='use crf')

parser.add\_argument('--mode', type=str, default='train',

help='train mode or test mode')

parser.add\_argument('--save', type=str, default='./checkpoints/lstm\_crf.pth',

help='path to save the final parameters of model')

parser.add\_argument('--save-model', type=str, default='./checkpoints/lstm\_crf\_model.pth',

help='path to save the final model')

parser.add\_argument('--save-epoch', action='store\_true',

help='save every epoch')

parser.add\_argument('--data', type=str, default='dataset',

help='location of the data corpus')

parser.add\_argument('--word-ebd-dim', type=int, default=100,

help='number of word embedding dimension')

parser.add\_argument('--dropout', type=float, default=0.5,

help='the probability for dropout')

parser.add\_argument('--lstm-hsz', type=int, default=512,

help='BiLSTM hidden size')

parser.add\_argument('--lstm-layers', type=int, default=2,

help='biLSTM layer numbers')

parser.add\_argument('--l2', type=float, default=0.005,

help='l2 regularization')

parser.add\_argument('--clip', type=float, default=2,

help='gradient clipping')

parser.add\_argument('--result-path', type=str, default='./result',

help='result-path')

parser.add\_argument('--vocab-path', type=str, default='./dataset/vocab.pkl',

help='vocab-path')

parser.add\_argument('--vec-path', type=str, default='./dataset/vec.txt',

help='vec-path')

args = parser.parse\_args()

torch.manual\_seed(args.seed)#设置随机数种子

args.device = torch.device("cuda" if args.use\_cuda else "cpu")#指定设备

3） 编写执行训练流程的函数train。该函数负责训练，接收一个参数epoch，表示轮次序号，函数主要工作如下：

• 设置model为训练模式；

• 遍历训练数据集；

• 获取一批数据后，首先梯度清零

• 将word和label传给model计算模型损失值

• 反向传播求导

• 优化器优化

• 打印输出训练信息

代码如下：

【代码30】main.py

def train(epoch):

model.train()#设置训练模式

total\_loss = 0

for word, label, \_ in train\_data:#遍历数据

optimizer.zero\_grad()#清零导数

loss, \_ = model(word, label)#计算损失

loss.backward()#反向传播

optimizer.step()#优化器优化

print("epoch:{},step:{}/{},loss:{}".format(epoch, train\_data.\_step, train\_data.num\_batchs, loss.item()))#打印信息

total\_loss += loss.detach()

return total\_loss / train\_data.num\_batchs #返回平均损失值

4）编写执行验证集验证的函数evaluate。该函数接收一个参数epoch，表示轮次序号。该函数流程如下：

• 根据tag2label（标注to id）得到label2tag（id to 标注）

• 调用模型进行预测，得到id序列并使用label2tag转换成tag序列

• 循环遍历预测结果和真实结果，组织成（真实文字，真实tag标注，预测tag标注）的形式

• 将上一步组织的结果写入文件

• 调用eval.py中的函数conlleval，通过调用perl脚本对预测结果进行指标统计，最终的到准确率、精确率、召回率，F1值以及各个标注类别的指标并输出

编写代码如下：

【代码34】main.py

def evaluate(epoch):

model.eval()

eval\_loss = 0

model\_predict = []

label2tag = {}

for tag, lb in tag2label.items():#根据tag2label得到label2tag（id to 标注）

label2tag[lb] = tag if lb != 0 else lb

label\_list = []

for word, label, seq\_len\_list in test\_data:#循环遍历测试数据

loss, \_ = model(word, label)#模型计算

pred = model.predict(word) # 模型预测

for i, seq\_len in enumerate(seq\_len\_list):

pred\_ = list(pred[i][:seq\_len].cpu().numpy())#获取预测id

label\_list.append(pred\_)

eval\_loss += loss.detach().item()

for label\_, (sent, tag) in zip(label\_list, data\_origin):

tag\_ = [label2tag[label\_\_] for label\_\_ in label\_]#id转换成tag

sent\_res = []

for i in range(len(sent)):

sent\_res.append([sent[i], tag[i], tag\_[i]])

model\_predict.append(sent\_res)

label\_path = os.path.join(args.result\_path, 'label\_' + str(epoch))#构建路径

metric\_path = os.path.join(args.result\_path, 'result\_metric\_' + str(epoch))

for line in conlleval(model\_predict, label\_path, metric\_path):#保存文件并统计

print(line)

return eval\_loss / test\_data.num\_batchs

5）编写执行训练和评估的主过程代码，该部分代码较长，我们分一下几个部分编写：

• 加载数据并实例化数据批次迭代器

• 加载预训练字向量和词表并实例化模型

• for循环遍历epochs执行train和evaluate

首先，我们加载数据并实例化训练数据迭代器和测试数据迭代器：

【代码36】main.py

# 加载数据

sents\_train, labels\_train,\_ = read\_corpus(os.path.join('.', args.data, 'train\_data.txt'),

os.path.join('.', args.data, 'train\_label.txt'))

sents\_test, labels\_test, data\_origin = read\_corpus(os.path.join('.', args.data, 'test\_data.txt'),

os.path.join('.', args.data, 'test\_label.txt'), is\_train=False)

args.label\_size = len(tag2label)#tag\_nums加入配置字典

train\_data = DataLoader(sents\_train,

labels\_train,

cuda=args.use\_cuda,

batch\_size=args.batch\_size)#训练集迭代器

test\_data = DataLoader(sents\_test,

labels\_test,

cuda=args.use\_cuda,

evaluation=True,

batch\_size=args.batch\_size)#测试集迭代器

然后，我们加载预训练字向量和词表，并存储到参数字典中，然后实例化并配置模型：

1-2行：加载字向量和词典

3-5行：实例化模型，然后加载已经训练过的模型参数

6行：实例化优化器

【代码37】main.py

#加载预训练词向量和词典

args.pre\_trained\_vec=get\_vector(args.vec\_path)

args.word2id=read\_dictionary(args.vocab\_path)

#实例化模型

model = Model(args)#实例化模型

model = model.to(args.device)

model.load\_state\_dict(torch.load(args.save,map\_location=args.device))#加载模型参数

#实例化优化器

optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)

最后，我们编写一个循环，遍历epochs，每次循环都会调用train函数实现一个epoch的迭代，然后调用evaluate函数进行评估，最终保存模型。

代码如下：

【代码37】main.py

if args.mode == 'train':

print('-' \* 90)

for epoch in range(1, args.epochs + 1):#遍历epochs

epoch\_start\_time = time.time()

loss = train(epoch)#训练

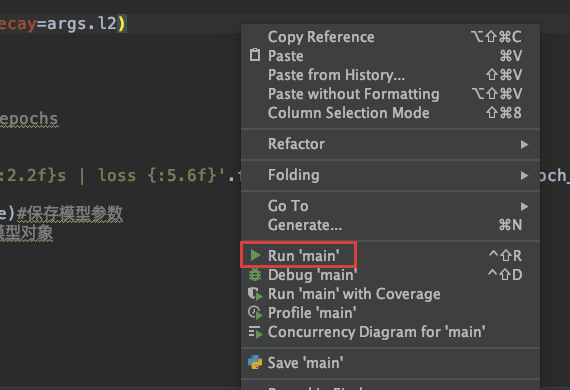
print('| start of epoch {:3d} | time: {:2.2f}s | loss {:5.6f}'.format(epoch, time.time() - epoch\_start\_time, loss))

eval\_loss = evaluate(epoch)#模型评估

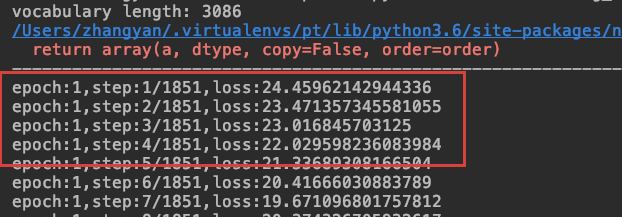
torch.save(model.state\_dict(), args.save)#保存模型参数

torch.save(model, args.save\_model)#保存模型对象

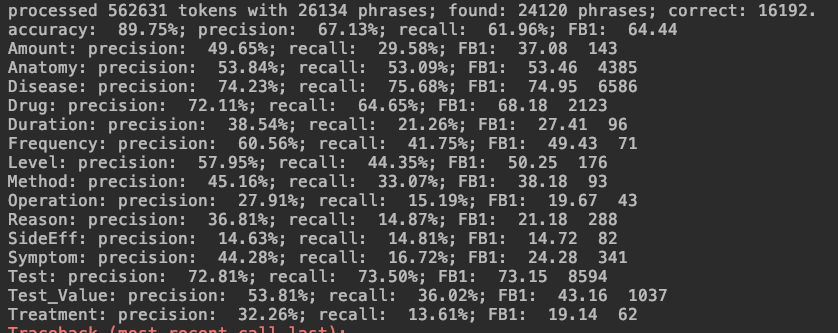
6）编写好代码后，我们可以开始训练了，在main.py文件中点击右键，选择“run main”：



运行后，稍等一会，在pycharm的下方就会输出如下内容：



这表示模型开始训练，我们等待模型训练，每训练一轮，都会输出在测试集上的评估结果，如下图所示：



7.6实现命名实体识别预测

模型训练完成后，我们使用训练好的模型来完成预测。整个过程如下：

• 给定模型路径和词典路径

• 加载词典和tag词典

• 给定要预测的文本并进行预处理

• 转成小写后拆成字，数字转换成<NUM>

• 用词典将每个字转换成对应的id，未知词转换成<UNK>

• 加载模型

• 模型预测

• 打印结果

在Medical-NER项目中新建Python File文件“predict”。

1）导入需要使用的包。

【代码37】predict.py

from data import tag2label,read\_dictionary

import torch

import numpy as np

2）继续编写如下代码：

【代码38】predict.py

model\_path='checkpoints/lstm\_crf\_model.pth'#模型路径

vocab\_path='dataset/vocab.pkl'#词典路径

word2id=read\_dictionary(vocab\_path)#加载词典

label2tag={v:k for k,v in tag2label.items()}#构建tag词典

text='随着生活水平的提高，2型糖尿病已经成为继心脑血管病、恶性肿瘤之后影响人类健康的第三大因素'

#数据处理

words=['<NUM>' if word.isdigit() else word for word in text.lower().strip() ]

words=[word2id[word] if word in word2id.keys() else word2id['<UNK>'] for word in words]

#加载模型

model = torch.load(model\_path)

#模型预测

\_,pred=model.predict(torch.from\_numpy(np.array([words])))

#按行打印结果

result=[[t,label2tag[id]] for t,id in zip(text.lower().strip(),list(pred.cpu().numpy()[0]))]

for l in result:

print(l)

其中:

第1-2行：指定模型路径和词典路径

第3行：加载词典

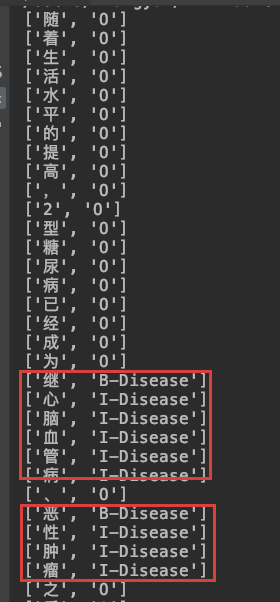
第4行：构建tag词典

第5-7行：给定数据并清洗

第8-9行：加载模型并预测

第10--13行：按行打印预测结果

3）运行predict.py文件，模型开始执行，并输入如下结果：



8.案例实验结果（结论）

1）模型训练。

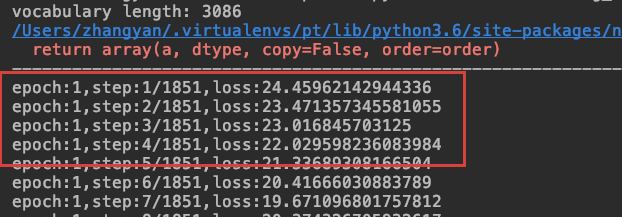
本次案例中，我们搭建了BiLstm+CRF模型，并且采用了高效的代码实现了CRF中的前向概率计算和viterbi解码算法，通过案例，最终的训练速度提高了50倍。目前案例中的模型效果还不是很好，还有很大的优化空间。可以考虑优化的方向有：

• 加入分词特征、词性特征、拼音特征等高级特征，这将能很大程度上提高模型效果。

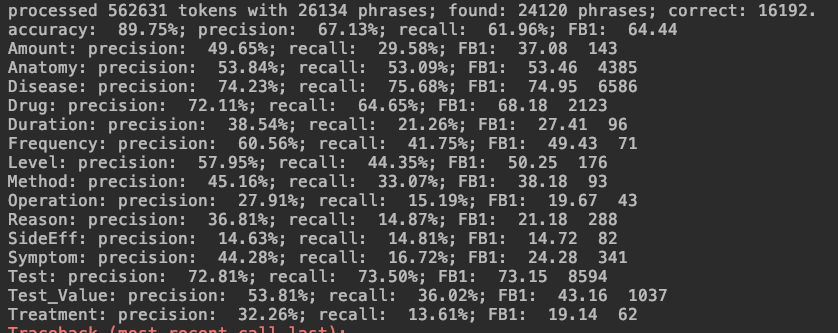
• 数据存在不平衡现象，可以尝试解决样本不平衡问题来提高模型效果；

• LSTM不适合并行运算，导致训练时间过长，也给模型调优带来困难，可以考虑使用IDCNN这样的基于CNN可并行化的模型来提高速度性能；

我们运行main.py文件，结果如下图所示：



每训练一个轮次，都会输出测试集评估结果，如下图所示：



上图中可以看到，模型的F1值为64.44，还需要继续训练，最终可以达到73-74的水平。对于命名实体识别任务来说，特别是这种实体种类偏多的情况，很难取得一个非常好的效果，这也是目前业界研究的一个方向。

2）模型预测。

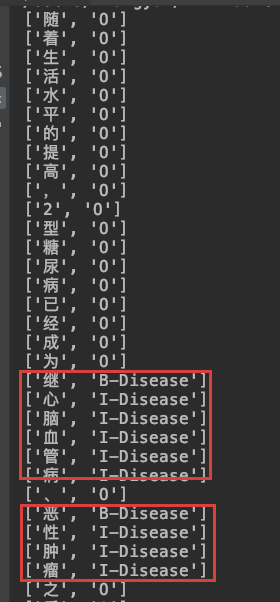
在模型训练的过程中，每训练一轮我们都保存了模型参数对象和模型对象本身，二者的区别为：

• 模型参数：只有参数，没有计算图

• 模型对象：序列化的模型对象，包含计算图和参数

我们实现的模型预测，就是直接导入了保存的模型对象，这样比较方便，无需我们再去使用大量参数实例化模型。

我们predict.py文件，结果如下图所示。



在上图中国可以看出，我们从“随着生活水平的提高，2型糖尿病已经成为继心脑血管病、恶性肿瘤之后影响人类健康的第三大因素”这句话中找到了2个“疾病”类型实体，分别为“继心脑血管病” 和“恶性肿瘤”，其中“继心脑血管病”多了前面的一个“继”，；另外，有一个实体“2型糖尿病”没有被识别出来。可以看出，模型还需要持续优化。

**9.案例代码**

【案例代码】data.py

import pickle, os

from scripts.const import \*

def read\_corpus(corpus\_path, label\_path, is\_train=True):

sents = []#保存所有句子

labels = []#保存所有标签

data = []#保存所有数据

with open(corpus\_path, encoding='utf-8') as fr:#打开文本数据文件

lines\_co = fr.readlines()#读取所有行

with open(label\_path) as fl:#打开标签文件

lines\_lb = fl.readlines()#读取所有行

if not is\_train:#如果不是训练，就加载已经创建好的词典文件

word2id = read\_dictionary('./dataset/vocab.pkl')

else:#如果是训练，则需要重新创建词典文件

word2id = {}

for line\_co, line\_lb in zip(lines\_co, lines\_lb):

sent\_,tag\_ = line\_co.strip().split(),line\_lb.strip().split()#去空格

if len(sent\_)==0:#句子长度为0 就不处理

continue

data.append((sent\_, tag\_))#原数据放入data

sentence\_id,label\_ = [],[]#保存词id构成的序列和标签id构成的序列

for word in sent\_:

word = word.lower()

if word.isdigit():#判断是数字就转换成<NUM>

word = '<NUM>'

if is\_train:

if word not in word2id:

word2id[word] = len(word2id)+1#加入词典

else:

if word not in word2id:#不再词典内就是未知词

word = '<UNK>'

sentence\_id.append(word2id[word])#转换成id加入列表

for tag in tag\_:

label = tag2label[tag]

label\_.append(label)

sents.append(sentence\_id)#保存到所有句子中

labels.append(label\_)#保存到所有标签序列中

if is\_train:

word2id['<UNK>'] = len(word2id)+1

word2id['<PAD>'] = PAD

print('vocabulary length:', len(word2id))

with open('./dataset/vocab.pkl', 'wb') as fw:

pickle.dump(word2id, fw)

return sents, labels, data

def read\_dictionary(vocab\_path):

vocab\_path = os.path.join(vocab\_path)#拼接路径

with open(vocab\_path, 'rb') as fr:

word2id = pickle.load(fr)#加载词典

return word2id#返回词典

def get\_vector(path):

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

vectors={}#空字典 用来保存读取的词向量

for line in f.readlines():

line=line.strip().split(' ')#空格分隔

word,vector = line[0],line[1:]#第一个是字，后面的是向量的值

vector=[float(i) for i in vector]#将数值字符串转换成小数

if len(vector)==100:#长度为100才对

vectors[word]=vector#放入字典

return vectors#返回

【案例代码】data\_loader.py

import numpy as np

import torch

from scripts.const import \*

class DataLoader(object):

def \_\_init\_\_(self, sents,labels, cuda=True,batch\_size=64, shuffle=False, evaluation=False):

self.cuda = cuda

self.sents=sents

self.labels=labels

self.evaluation = evaluation

self.\_batch\_size = batch\_size

if not evaluation:

self.\_augment()#实现数据增强

self.sents,self.labels = np.asarray(self.sents),np.asarray(self.labels)

if not evaluation:

self.\_clean()#实现数据清理

self.sents\_size = len(self.sents)

self.\_step = 0#用来记录迭代时的批次下标

self.num\_batchs = self.sents\_size // batch\_size#计算批次数量

self.divided=True#记录是否整除

if self.sents\_size%batch\_size!=0:#不能整除

self.divided=False

self.num\_batchs+=1#批次数量+1

if shuffle:

self.\_shuffle()#打乱数据

elif not evaluation:

self.\_sorted()#排序数据

def \_augment(self):

#前后两句拼接成一个句子

self.sents=self.sents+[self.sents[i]+self.sents[i+1] for i in range(len(self.sents)-1)]

self.labels = self.labels + [self.labels[i] + self.labels[i + 1] for i in range(len(self.labels) - 1)]

def \_clean(self):

#删除标签全部是O的句子,就是没有实体的句子

sums=[len(set(item))>1 for item in self.labels]#计算句子中非O的个数

indices=np.where(sums)[0]#筛选出全是O的句子

self.sents=self.sents[indices]#布尔索引实现文本筛选

self.labels = self.labels[indices]#对应的标签也要去掉

def \_shuffle(self):

indices=np.random.permutation(self.sents\_size)#获取乱序下标

self.sents = self.sents[indices]#文本数据重新索引

self.labels = self.labels[indices]#标签数据重新索引

def \_sorted(self):

lens=np.array([len(sent) for sent in self.sents])#计算句子长度

indices=np.argsort(lens)#排序

self.sents = self.sents[indices]#重新索引

self.labels = self.labels[indices]#重新索引

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

return self

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

if self.\_step == self.num\_batchs:

self.\_step = 0

raise StopIteration()

\_start = self.\_step \* self.\_batch\_size

if (self.\_step == self.num\_batchs-1 and self.divided==False):

\_sp=self.sents\_size%self.\_batch\_size

else:

\_sp=self.\_batch\_size

self.\_step += 1

if not self.evaluation:

indices = np.random.permutation(\_sp)

else:

indices=np.arange(\_sp)

word, seq\_len\_list = self.\_pad\_to\_longest(self.sents[\_start:\_start + \_sp][indices])

label, \_ = self.\_pad\_to\_longest(self.labels[\_start:\_start + \_sp][indices])

return word, label, seq\_len\_list

def \_pad\_to\_longest(self,insts):

seq\_len\_list = [len(inst) for inst in insts]

max\_len = max(len(inst) for inst in insts)

inst\_data = np.array(

[inst + [PAD] \* (max\_len - len(inst)) for inst in insts])

if self.evaluation:

with torch.no\_grad():

inst\_data\_tensor = torch.from\_numpy(inst\_data)

else:

inst\_data\_tensor = torch.from\_numpy(inst\_data)

if self.cuda:

inst\_data\_tensor = inst\_data\_tensor.cuda()

return inst\_data\_tensor, seq\_len\_list

【案例代码】model.py

import torch

import torch.nn as nn

from torch.nn import init

import numpy as np

from scripts.const import \*

class CRF(nn.Module):

def \_\_init\_\_(self, label\_size, device):

super().\_\_init\_\_()

self.label\_size = label\_size

self.device=device

self.transitions = nn.Parameter(

torch.randn(label\_size, label\_size)) # 定义转移矩阵

init.xavier\_uniform\_(self.transitions) # 初始化

self.transitions.data[START, :].fill\_(-10000.) # 起始行全部设为-1000

self.transitions.data[:, STOP].fill\_(-10000.) # 终止列全部设为-1000

def \_log\_sum\_exp(self,input, keepdim=False):

assert input.dim() == 2

max\_scores, \_ = input.max(dim=-1, keepdim=True)# max\_score维度是１

output = input - max\_scores#减去最大值，谜底是防止指数运算上溢

return max\_scores + torch.log(torch.sum(torch.exp(output), dim=-1, keepdim=keepdim))

def \_score\_sentence(self, feats, tags):

score = torch.zeros(tags.shape[0]).to(self.device)

s\_score = torch.LongTensor([[START]] \* tags.shape[0]).to(self.device)#初始化分值

tags = torch.cat([s\_score, tags], dim=-1)

for i in range(feats.shape[1]):#遍历每一个token

feat=feats[:,i,:]#获取当前token的发射概率

#加上转移概率和发射概率

score = score + self.transitions[tags[:,i + 1], tags[:,i]] + feat[range(feat.shape[0]),tags[:,i + 1]]

score = score + self.transitions[STOP, tags[:,-1]]

return score

def forward(self,input):

# 初始化路径分值

init\_vvars = torch.full((1, self.label\_size), -10000.).to(self.device)

init\_vvars[0][START] = 0

forward\_var\_list = []

forward\_var\_list.append(torch.stack([init\_vvars] \* input.shape[0]).squeeze(1)) # 第一个时刻

for feat\_index in range(input.shape[1]):

gamar\_r\_l = forward\_var\_list[feat\_index].unsqueeze(1) # [128,1,33]

next\_tag\_var = gamar\_r\_l + self.transitions # 128,33,33 加上转移概率

t\_r1\_k = torch.unsqueeze(input[:, feat\_index, :], -1) # 128,33,1 发射概率

forward\_var\_new = next\_tag\_var + t\_r1\_k # 128,33,33 加上发射概率

forward\_var\_list.append(torch.logsumexp(forward\_var\_new, dim=2))#求和

terminal\_var = forward\_var\_list[-1] + self.transitions[STOP].repeat([input.shape[0], 1])

return torch.logsumexp(terminal\_var, dim=1)

def \_viterbi\_decode(self, feats):

backpointers = []

#初始化路径分值

init\_vvars = torch.full((1, self.label\_size), -10000.).to(self.device)

init\_vvars[0][START] = 0

forward\_var\_list = []

forward\_var\_list.append(torch.stack([init\_vvars] \* feats.shape[0]))#第一个时刻

for feat\_index in range(feats.shape[1]):

gamar\_r\_l = forward\_var\_list[feat\_index]#[128,1,33]

next\_tag\_var = gamar\_r\_l + self.transitions#128,33,33 加上转移概率

t\_r1\_k = torch.unsqueeze(feats[:,feat\_index,:], -1)#128,33,1 发射概率

forward\_var\_new = next\_tag\_var + t\_r1\_k#128,33,33 加上发射概率

#取出当前时刻的最大分值状态集状态下标

viterbivars\_t, bptrs\_t = torch.max(forward\_var\_new, dim=-1) # 128,33 128,33

forward\_var\_list.append(viterbivars\_t.unsqueeze(1))

backpointers.append(bptrs\_t)

#转移到最后的stop

terminal\_var = forward\_var\_list[-1] + self.transitions[STOP]

# 找到最大路径终止时刻tag下标

path\_score,best\_tag\_id = torch.max(terminal\_var.squeeze(1), dim=-1)

#回溯解码

best\_tag\_id=best\_tag\_id.unsqueeze(1)

best\_path = [best\_tag\_id]

for bptrs\_t in reversed(backpointers):

best\_tag\_id = torch.gather(bptrs\_t,1,best\_tag\_id)#获取上一时刻路径状态

best\_path.append(best\_tag\_id)

best\_path.pop()#START不要

best\_path.reverse()#逆转

return path\_score, torch.cat(best\_path, dim=-1)

class BiLSTM(nn.Module):

def \_\_init\_\_(self, word\_ebd\_dim, lstm\_hsz, lstm\_layers, dropout, batch\_size,vec,word2id,device):

super().\_\_init\_\_()

self.lstm\_layers = lstm\_layers

self.lstm\_hsz = lstm\_hsz

self.batch\_size = batch\_size

self.word2id=word2id

self.word\_size=len(word2id)

self.word\_ebd\_dim=word\_ebd\_dim

self.device=device

self.scope=1

self.vec=vec

if vec:#如果预训练字向量存在

vecs =self.\_get\_vec()#获取字向量

self.word\_ebd = nn.Embedding.from\_pretrained(vecs)#使用预训练的字向量实例化嵌入层

else:

self.word\_ebd = nn.Embedding(self.word\_size, self.word\_ebd\_dim)#随机初始化嵌入层

self.word\_ebd.weight.data.uniform\_(self.scope, self.scope)

self.lstm = nn.LSTM(word\_ebd\_dim,

hidden\_size=lstm\_hsz // 2,

num\_layers=lstm\_layers,

batch\_first=True,

dropout=dropout,

bidirectional=True)#实现BiLstm层

def \_get\_vec(self):

weight = torch.Tensor(self.word\_size, self.word\_ebd\_dim).uniform\_(-self.scope,self.scope)

for k,v in self.vec.items():

if k in self.word2id.keys():

index = self.word2id[k]#获取词对应的id

weight[index, :] = torch.from\_numpy(np.array(v))#将该词对应的嵌入向量修改为预训练向量

return weight

def forward(self, words):

encode = self.word\_ebd(words)#嵌入计算

#初始化状态

hidden=self.init\_hidden(words.shape[0])

output, hidden = self.lstm(encode,hidden)#sltm计算

return output, hidden

def init\_hidden(self,batch\_size):

return (torch.zeros(self.lstm\_layers \* 2, batch\_size, self.lstm\_hsz // 2).to(self.device),

torch.zeros(self.lstm\_layers \* 2, batch\_size, self.lstm\_hsz // 2).to(self.device))

class Model(nn.Module):

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

super().\_\_init\_\_()

for k, v in args.\_\_dict\_\_.items():#参数赋值

self.\_\_setattr\_\_(k, v)

self.bilstm = BiLSTM(self.word\_ebd\_dim,

self.lstm\_hsz, self.lstm\_layers,

self.dropout, self.batch\_size,

self.pre\_trained\_vec,self.word2id,self.device

)#bilstm层

self.logistic = nn.Linear(self.lstm\_hsz, self.label\_size)#Linear层

self.crf = CRF(self.label\_size, self.device)#crf层

torch.nn.init.xavier\_normal\_(self.logistic.weight)#初始化linear参数

self.logistic.bias.data.fill\_(0)

def forward(self, words, labels, hidden=None):

output, \_ = self.bilstm(words)#bilstm计算

output = self.logistic(output)#linear层计算

pre\_score = self.crf.forward(output)#所有路径分值和

label\_score = self.crf.\_score\_sentence(output, labels)#

return (pre\_score - label\_score).mean(), None

def predict(self, word):

lstm\_out, \_ = self.bilstm(word)

out = self.logistic(lstm\_out)

return self.crf.\_viterbi\_decode(out)

【案例代码】main.py

import argparse

import torch

from model import Model

from data\_loader import DataLoader

from data import read\_corpus, tag2label,read\_dictionary,get\_vector

import os

from scripts.eval import conlleval

import time

parser = argparse.ArgumentParser(description='LSTM\_CRF')

parser.add\_argument('--epochs', type=int, default=100,

help='number of epochs for train')

parser.add\_argument('--batch-size', type=int, default=128,

help='batch size for training')

parser.add\_argument('--test-size', type=int, default=128,

help='batch size for testing')

parser.add\_argument('--seed', type=int, default=1111,

help='random seed')

parser.add\_argument('--use-cuda', action='store\_true',default=False,

help='enables cuda')

parser.add\_argument('--lr', type=float, default=0.005,

help='learning rate')

parser.add\_argument('--use-crf', action='store\_true',

help='use crf')

parser.add\_argument('--mode', type=str, default='train',

help='train mode or test mode')

parser.add\_argument('--save', type=str, default='./checkpoints/lstm\_crf.pth',

help='path to save the final parameters of model')

parser.add\_argument('--save-model', type=str, default='./checkpoints/lstm\_crf\_model.pth',

help='path to save the final model')

parser.add\_argument('--save-epoch', action='store\_true',

help='save every epoch')

parser.add\_argument('--data', type=str, default='dataset',

help='location of the data corpus')

parser.add\_argument('--word-ebd-dim', type=int, default=100,

help='number of word embedding dimension')

parser.add\_argument('--dropout', type=float, default=0.5,

help='the probability for dropout')

parser.add\_argument('--lstm-hsz', type=int, default=256,

help='BiLSTM hidden size')

parser.add\_argument('--lstm-layers', type=int, default=2,

help='biLSTM layer numbers')

parser.add\_argument('--l2', type=float, default=0.002,

help='l2 regularization')

parser.add\_argument('--clip', type=float, default=5,

help='gradient clipping')

parser.add\_argument('--result-path', type=str, default='./result',

help='result-path')

parser.add\_argument('--vocab-path', type=str, default='./dataset/vocab.pkl',

help='vocab-path')

parser.add\_argument('--vec-path', type=str, default='./dataset/vec.txt',

help='vec-path')

args = parser.parse\_args()

torch.manual\_seed(args.seed)

args.device = torch.device("cuda" if args.use\_cuda else "cpu")

def train(epoch):

model.train()

total\_loss = 0

for word, label, \_ in train\_data:

optimizer.zero\_grad()

loss, \_ = model(word, label)

loss.backward()

optimizer.step()

print("epoch:{},step:{}/{},loss:{}".format(epoch, train\_data.\_step, train\_data.num\_batchs, loss.item()))

total\_loss += loss.detach()

return total\_loss / train\_data.num\_batchs

def evaluate(epoch):

model.eval()

eval\_loss = 0

model\_predict = []

label2tag = {}

for tag, lb in tag2label.items():

label2tag[lb] = tag if lb!=0 else lb

label\_list = []

with torch.no\_grad():

for word, label, seq\_len\_list in test\_data:

loss, \_ = model(word, label)

\_,pred = model.predict(word) # 模型预测

for i, seq\_len in enumerate(seq\_len\_list):

pred\_ = list(pred[i][:seq\_len].cpu().numpy())

label\_list.append(pred\_)

eval\_loss += loss.detach().item()

for label\_, (sent, tag) in zip(label\_list, data\_origin):

tag\_ = [label2tag[label\_\_] for label\_\_ in label\_]

sent\_res = []

if len(label\_) != len(sent):

print(sent)

print(len(label\_))

print(tag)

for i in range(len(sent)):

sent\_res.append([sent[i], tag[i], tag\_[i]])

model\_predict.append(sent\_res)

label\_path = os.path.join(args.result\_path, 'label\_' + str(epoch))

metric\_path = os.path.join(args.result\_path, 'result\_metric\_' + str(epoch))

for line in conlleval(model\_predict, label\_path, metric\_path):

print(line)

return eval\_loss / test\_data.num\_batchs

# 加载数据

sents\_train, labels\_train,\_ = read\_corpus(os.path.join('.', args.data, 'train\_data.txt'), os.path.join('.', args.data, 'train\_label.txt'))

sents\_test, labels\_test, data\_origin = read\_corpus(os.path.join('.', args.data, 'test\_data.txt'), os.path.join('.', args.data, 'test\_label.txt'), is\_train=False)

args.label\_size = len(tag2label)

train\_data = DataLoader(sents\_train, labels\_train, cuda=args.use\_cuda, batch\_size=args.batch\_size)

test\_data = DataLoader(sents\_test, labels\_test, cuda=args.use\_cuda, evaluation=True, batch\_size=args.test\_size)

#加载预训练词向量和词典

args.pre\_trained\_vec=get\_vector(args.vec\_path)

args.word2id=read\_dictionary(args.vocab\_path)

#实例化模型

model = Model(args)

model = model.to(args.device)

model.load\_state\_dict(torch.load(args.save,map\_location=args.device))

optimizer = torch.optim.Adam(model.parameters(), lr=args.lr) #betas=(0.9, 0.98), eps=1e-9, weight\_decay=args.l2)

train\_loss = []

if args.mode == 'train':

print('-' \* 90)

for epoch in range(1, args.epochs + 1):

epoch\_start\_time = time.time()

loss = train(epoch)#训练

print('| start of epoch {:3d} | time: {:2.2f}s | loss {:5.6f}'.format(epoch, time.time() - epoch\_start\_time, loss))

eval\_loss = evaluate(epoch)#模型评估

torch.save(model.state\_dict(), args.save)#保存模型参数

torch.save(model, args.save\_model)#保存模型对象

【案例代码】predict.py

from data import tag2label,read\_dictionary

import torch

import numpy as np

model\_path='checkpoints/lstm\_crf\_model.pth'#模型路径

vocab\_path='dataset/vocab.pkl'#词典路径

word2id=read\_dictionary(vocab\_path)#加载词典

label2tag={v:k for k,v in tag2label.items()}#构建tag词典

text='随着生活水平的提高，2型糖尿病已经成为继心脑血管病、恶性肿瘤之后影响人类健康的第三大因素'

#数据处理

words=['<NUM>' if word.isdigit() else word for word in text.lower().strip() ]

words=[word2id[word] if word in word2id.keys() else word2id['<UNK>'] for word in words]

#加载模型

model = torch.load(model\_path)

#模型预测

\_,pred=model.predict(torch.from\_numpy(np.array([words])))

#按行打印结果

result=[[t,label2tag[id]] for t,id in zip(text.lower().strip(),list(pred.cpu().numpy()[0]))]

for l in result:

print(l)