

# 一、背景

## 1. 命名实体识别概述

知识抽取是实现自动化构建大规模知识图谱的重要技术，其目的在于从不同来源、不同结构的数据中进行知识提取并存入知识图谱中（详情参考[知识图谱 - Tintoki\\_blog\\_\(gintoki-jpg.github.io\)](#)）；知识抽取同样属于NLP的研究领域，指自动化地从文本中发现和抽取相关信息，并将多个文本碎片中的信息进行合并，将非结构化数据转换为结构化数据；

非结构化数据的知识抽取包括命名实体识别、关系抽取及事件抽取，本项目主要针对其中的命名实体识别；

命名实体识别是指从文本中检测出命名实体，并将其分类到预定义的类别中，例如人物、组织、**地点**、**时间**等；一般情况下，命名实体识别是知识抽取其他任务的基础；

想要从文本中进行实体抽取，首先需要从文本中识别和定位实体，然后再将识别的实体分类到预定义的类别中去 -- 这也是本项目需要实现的，即实体的识别和抽取；

实体抽取的方法分为三类：基于规则的方法、基于统计模型的方法和基于深度学习的方法，因为规定使用深度学习，所以我们这里仅介绍该方法，另外两类可参考[小知识 | 知识图谱：知识抽取之命名实体 \(qq.com\)](#)；

### 1.1 基于深度学习的方法

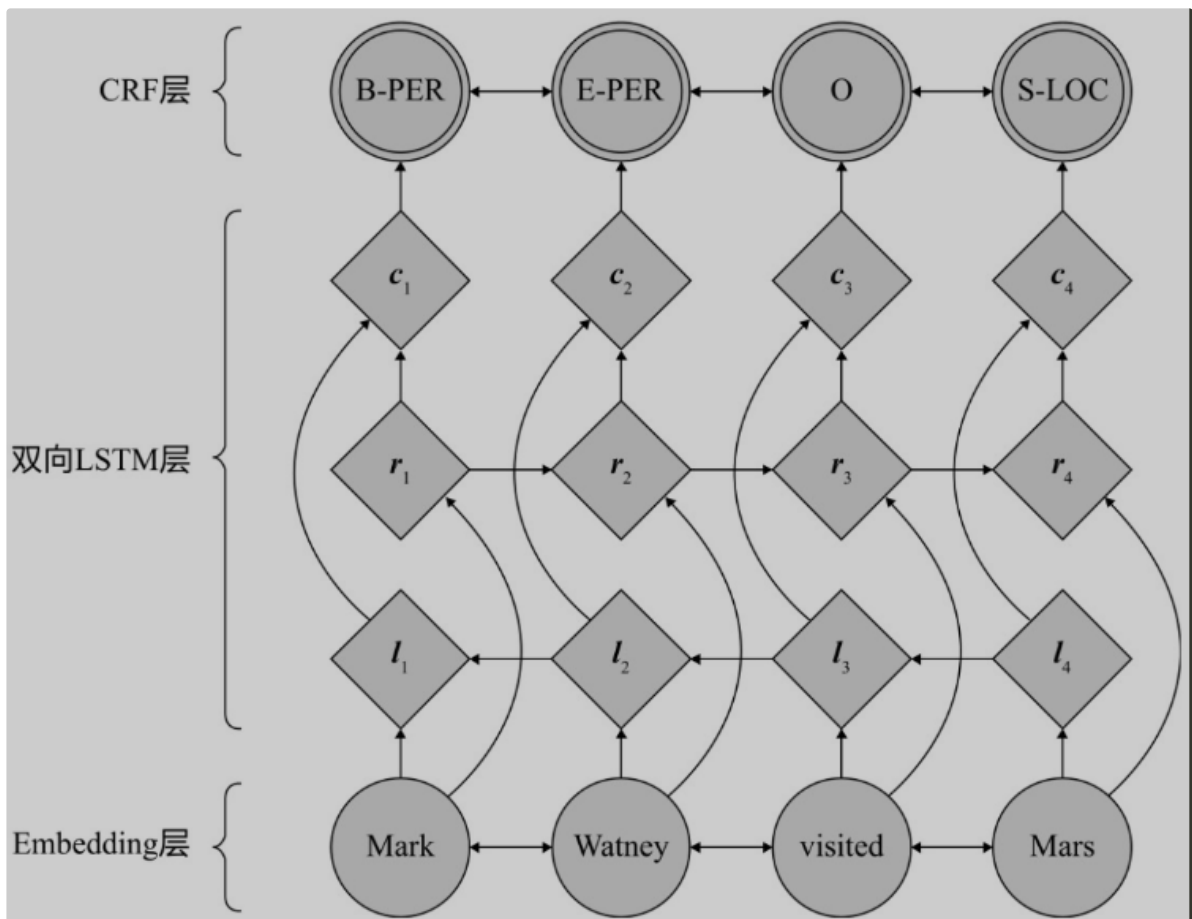
与传统统计模型相比，基于深度学习的方法直接以文本中词的向量为输入，通过神经网络实现端到端的命名实体识别，不再依赖人工定义的特征；

目前，用于命名实体识别的神经网络主要有卷积神经网络（Convolutional Neural Network, CNN）、循环神经网络（Recurrent Neural Network, RNN）以及引入注意力机制（Attention Mechanism）的神经网络；一般地，不同的神经网络结构在命名实体识别过程中扮演编码器的角色，它们基于初始输入以及词的上下文信息，得到每个词的新向量表示；最后再通过CRF模型输出对每个词的标注结果；

一些比较经典的用于实体识别抽取的模型有

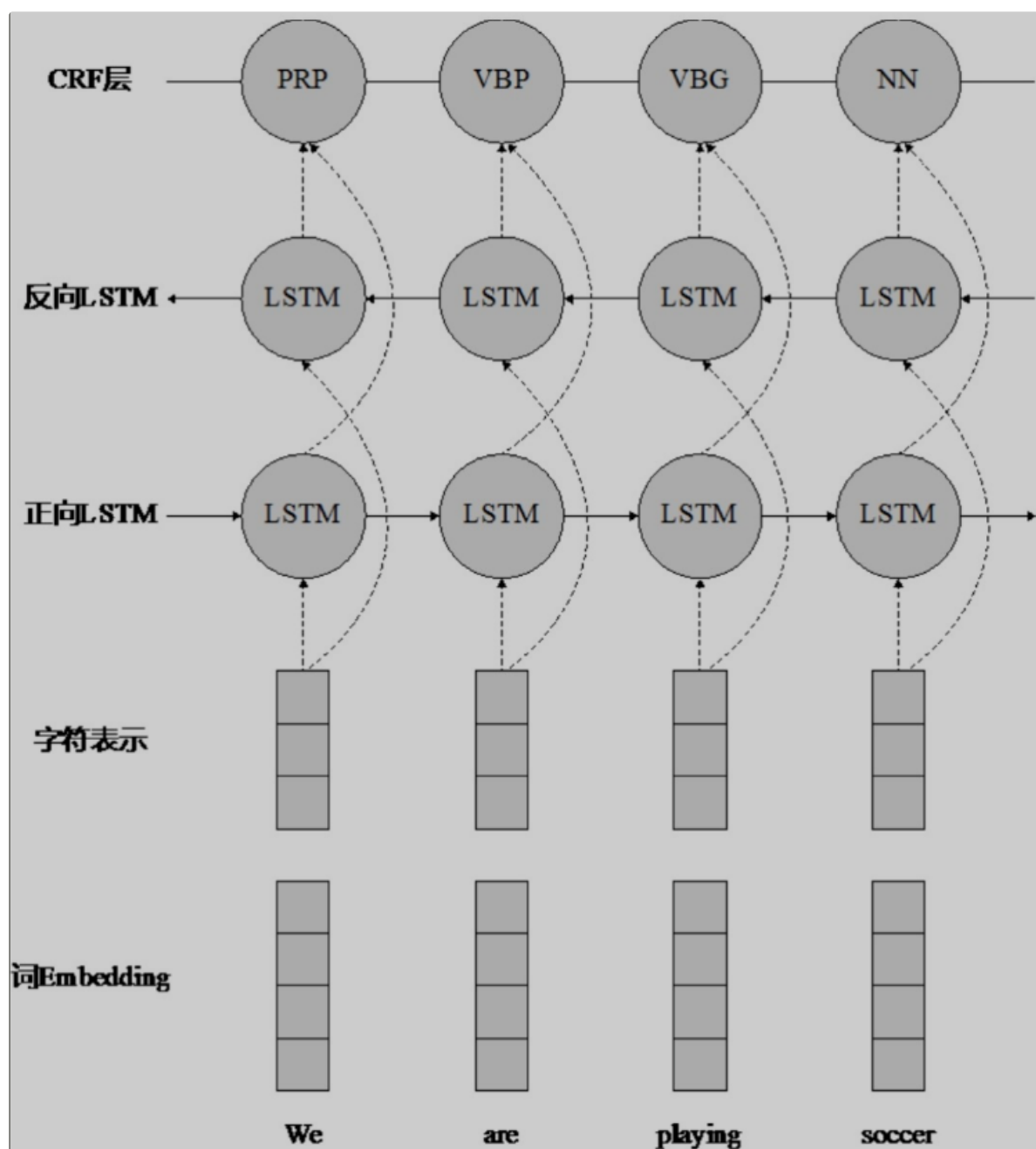
#### LSTM-CRF命名实体识别模型

该模型使用了长短期记忆神经网络（Long Short-Term Memory Neural Network, LSTM）与CRF相结合进行命名实体识别。



LSTM-CNNs-CRF序列标注模型框架

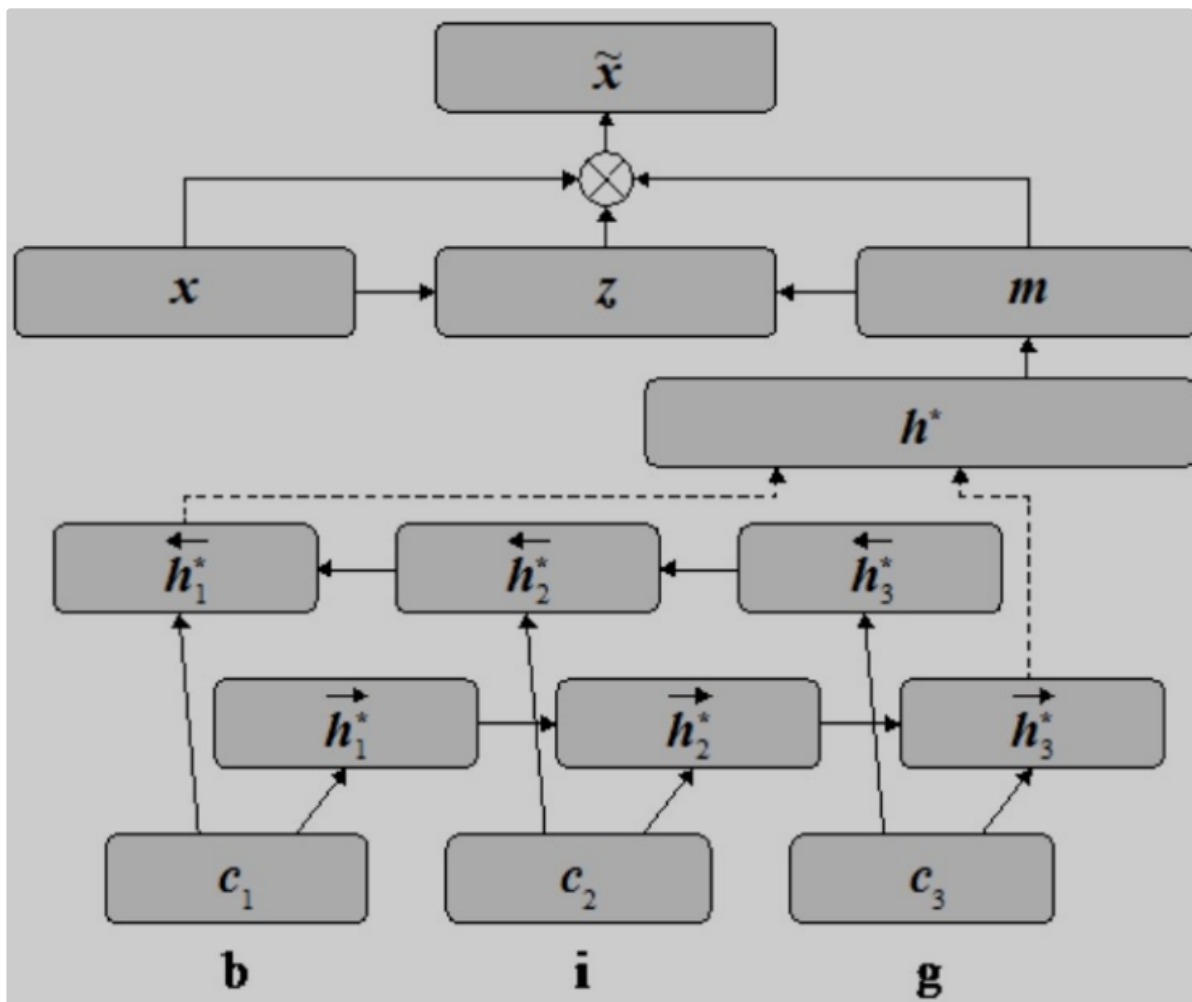
该模型与 LSTM-CRF 模型十分相似，不同之处是在Embedding 层中加入了每个词的字符级向量表示。



#### 基于注意力机制的词向量和字符级向量组合方法

注意力机制可以帮助扩展基本的编码器-解码器模型结构，让模型能够获取输入序列中与下一个目标词相关的信息。

基于注意力机制的词向量和字符级向量组合方法认为除了将词作为句子基本元素学习得到的特征向量，命名实体识别还需要词中的字符级信息。因此，该方法除了使用双向 LSTM 得到词的特征向量，还基于双向LSTM计算词的字符级特征向量。



## 2. 中文命名实体识别

### 2.1 中文分词

中文命名实体抽取需要先了解基于字标注的中文分词，简单的中文分词我们知道形式如下

"我爱北京天安门"。

基于字标注的中文分词结果如下

"我/O 爱/O 北/B 京/E 天/B 安/M 门/E"。

基于字标注的意思就是给每个字都进行标注，上述标注的类型主要有四种

B | 词首

M | 词中

E | 词尾

O | 单字

词首即一个词的开始，词尾即一个词的结束，词中表示词中间的词，假如该词只有一个字则用单字表示；

## 2.2 数据处理

实体识别和中文分词类似，就是将不属于实体的字用O标注，把实体用BME规则标注，最后按照BME规则将实体提取出来即可；

下面是一个实体识别的例子

```
{productname:浙江在线杭州}{time:4 月 25 日}讯 (记者{personname: 施宇翔} 通讯员  
{personname:方英}) 毒贩很“时髦”，用{productname:微信}交易毒品。没料想警方也很  
“潮”，将计就计，一举将其擒获。
```

每个实体用都用大括号括了起来，并标明实体类别（标注方式并不需要严格遵守这样的键值格式，只要能将实体识别并提取出来即可）

因为下面的例子都是基于玻森数据提供的命名实体识别数据，与我们项目本身提供的数据集可能存在一些差别，所以先简单介绍一下玻森数据集，主要包含以下6个实体类别

time: 时间

location: 地点

personname: 人名

orgname: 组织名

companyname: 公司名

productname: 产品名

数据处理首先要做的就是将原始数据按照BME0的规则变成字标注的形式便于模型训练，上述文本按字标注后的结果如下（可以看出这不仅仅是简单的字标注分词，同时结合了实体类别）

浙/Bproductname 江/Mproductname 在/Mproductname 线/Mproductname  
杭/Mproductname 州/Eproductname 4/Btime 月/Mtime 2/Mtime 5/Mtime 日/Etime 讯/O  
(/O 记/O 者/O /Bpersonname 施/Mpersonname 宇/Mpersonname 翔/Epersonname /O  
通/O 讯/O 员/O /O 方/Bpersonname 英/Epersonname ) /O 毒/O 贩/O 很/O “/O 时/O  
髦/O ” /O , /O 用/O 微/Bproductname 信/Eproduct\_name 交/O 易/O 毒/O 品/O 。/O  
没/O 料/O 想/O 警/O 方/O 也/O 很/O “/O 潮/O ” /O , /O 将/O 计/O 就/O 计/O , /O 一/O  
举/O 将/O 其/O 擒/O 获/O 。

接着习惯性的，按照标点符号将一个长句子分为多个短句子（逗号、句号、双引号等），结果如下

浙/Bproductname 江/Mproductname 在/Mproductname 线/Mproductname  
杭/Mproductname 州/Eproductname 4/Btime 月/Mtime 2/Mtime 5/Mtime 日/Etime 讯/O  
记/O 者/O /Bpersonname 施/Mpersonname 宇/Mpersonname 翔/Epersonname /O 通/O  
讯/O 员/O /O 方/Bpersonname 英/Epersonname  
毒/O 贩/O 很/O  
时/O 髦/O  
用/O 微/Bproductname 信/Eproduct\_name 交/O 易/O 毒/O 品/O  
没/O 料/O 想/O 警/O 方/O 也/O 很/O  
潮/O  
将/O 计/O 就/O 计/O  
一/O 举/O 将/O 其/O 擒/O 获/O

与先前新闻文本分类相同，因为无法直接将文本类型的数据放入模型训练，因此需要先建立一个word2id词典，将每个汉字转换成id（最直观的做法就是按照数据集中中汉字出现的次数进行排序后赋id，id从1开始）

的	1
1	2
0	3
	4
2	5
在	6
中	7
国	8
年	9
一	10
了	11
日	12
是	13
月	14
大	15
为	16
人	17
3	18
上	19
有	20
5	21
行	22

将汉字转换为id后，再建立一个tag2id词典，将每个字标注的类型转换成id（这里的id从1开始，顺序可以自定义，因为3\*6+1所以一共19个tag对应的id）

B_product_name	1
B_org_name	2
M_org_name	3
E_time	4
M_product_name	5
M_person_name	6
E_location	7
M_time	8
O	9
E_product_name	10
B_time	11
E_person_name	12
E_company_name	13
M_company_name	14
M_location	15
B_company_name	16
B_person_name	17
B_location	18
E_org_name	19
dtype: int64	

拥有了word2id和tsg2id之后，就可以将先前按照标点切分的短句以——对应的顺序将汉字和每个字的标签转换为id，分别存放在两个数组中，将该数组保存在同一个pkl文件中，这样模型使用时就可以直接读取，不用每次都处理数据了；

这里习惯把每一句话都转换成一样的长度（这与先前新闻文本分类是一个道理），这个长度可以自定义（最好统计后再确定），比它长的就把后面舍弃，比它短的就后面补零。

比如下面是长度为10的文本对应的word2id和tag2id

```
[132,45,0,456,432,8,654,3,0,0]  
[1,2,2,2,3,4,5,5,0,0]
```

Q：为什么这里是按照字的粒度而不是词的粒度划分？词向量和字向量的区别在哪里？

A：词向量和字向量都是自然语言处理中常用的表示文本的方式，但它们的表示粒度不同。

词向量 (Word Embedding) 是将每个单词表示为一个向量（因此使用词向量之前需要进行分词），这个向量通常是一个固定长度的实数向量，每个维度代表着该单词在不同语义维度上的分布情况。词向量的好处是能够捕捉到单词之间的语义关系，例如在词向量空间中，语义相近的单词的向量距离较近。常用的词向量算法有 word2vec、GloVe 等。

字向量 (Character Embedding) 则是将每个字母或字符表示为一个向量。相比于词向量，字向量的表示粒度更细，可以更好地捕捉词语的构成和形态等信息，特别适用于中文和其他一些没有明确词汇边界的语言。常用的字向量算法有 FastText、CharCNN 等。

因此，词向量适用于处理基于单词的任务，如文本分类、情感分析、机器翻译等，而字向量适用于处理基于字符的任务，如中文分词、命名实体识别等。

综上，对于本次任务使用字向量更合适。

### 3. BiLSTM-CRF 介绍

参考链接：[彻底了解 BiLSTM 和 CRF 算法-pytorch bilstm crf \(51cto.com\)](#);

BiLSTM-CRF 是一种序列标注模型，常用于自然语言处理领域的命名实体识别、词性标注等任务。其全称为 Bidirectional Long Short-Term Memory - Conditional Random Field，结合了双向长短时记忆网络 (Bidirectional LSTM) 和条件随机场 (CRF) 两个模型的优点，能够克服单向 LSTM 模型无法处理双向上下文信息的问题，同时能够利用 CRF 模型的全局标注优化策略来提高模型的准确性。

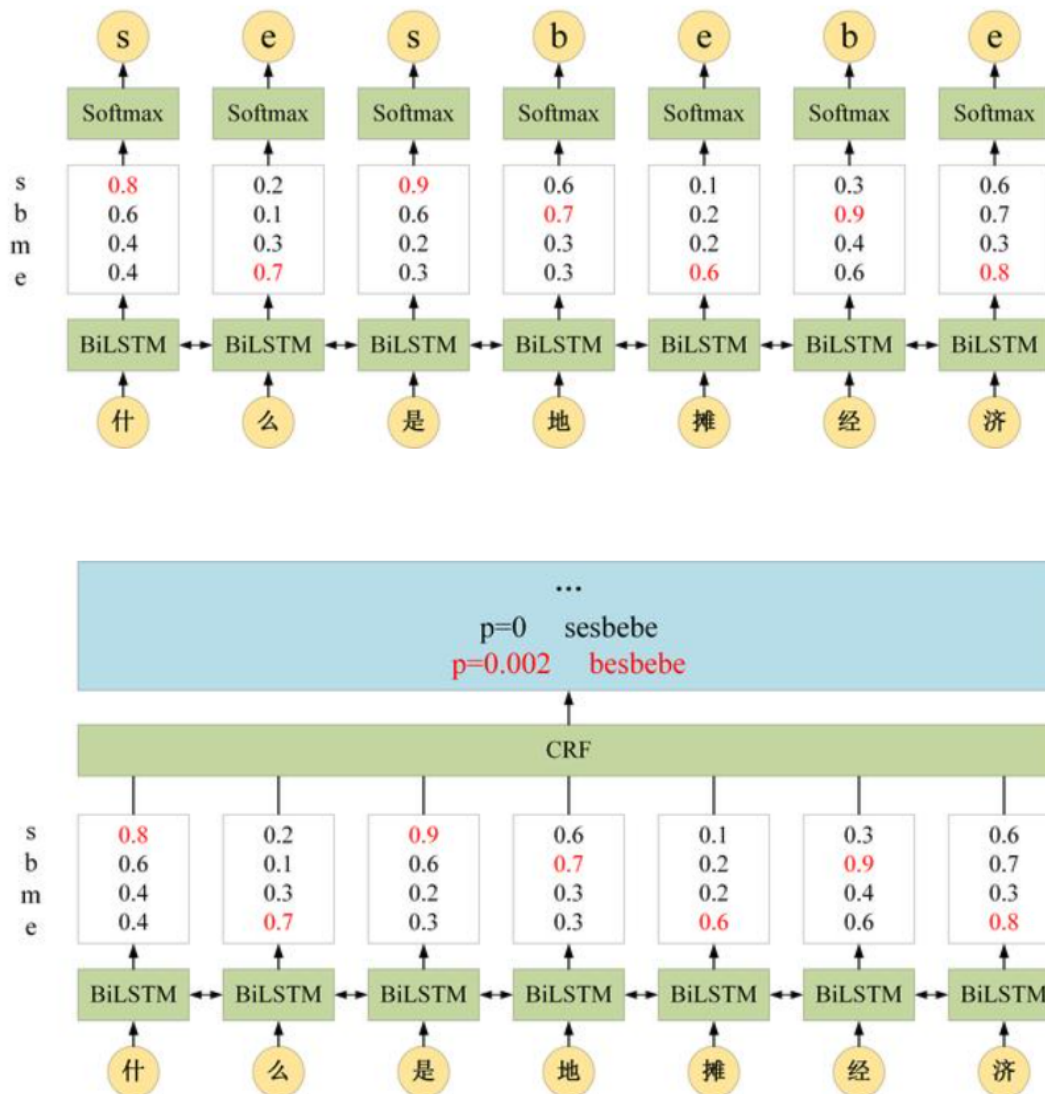
BiLSTM 模型是一种递归神经网络，它可以学习长文本序列中的特征，具有前向和后向两个方向的传播。与传统的单向 LSTM 模型相比，它能够捕捉到上下文中的更多信息，有利于提高模型的准确性。而 CRF 模型则是一种概率图模型，能够通过考虑全局标注的约束条件来优化模型的输出结果，进一步提高模型的准确性。

在 BiLSTM-CRF 模型中，BiLSTM 用于学习上下文特征，将上下文特征序列作为 CRF 的输入，CRF 则用于学习标签之间的转移概率，从而能够更好地对标注序列进行建模，从而实现更准确的序列标注任务。

为什么不单独使用 BiLSTM 进行标注？BiLSTM 可以预测出每一个字属于不同标签的概率，然后使用 Softmax 得到概率最大的标签，作为该位置的预测值。这样在预测的时候会忽略了标签之间的关联性，例如 BiLSTM 在作分词任务时，将某句话的第一个词预测为动词，紧接着的第二个动词同样被预测为动词，而实际上动词后面不能直接跟动词，因此 BiLSTM 没有考虑标签间联系。此时需要在 BiLSTM 的输出层加上一个 CRF，使得模型可以考虑类标之间的相关性，标签之间的相关性就是 CRF 中的转移矩阵，表示从一个状态转移到另一个状态的概率。

参考如下分词任务





综上，BiLSTM+CRF 考虑的是整个类标路径的概率而不仅仅是单个类标的概率。

### 3.1 CRF特征函数

CRF包含两种特征函数，第一种特征函数是状态特征函数，也称为发射概率，表示字  $x$  对应标签  $y$  的概率

$$\mu_l s_l(y_i, x, i) \quad l = 1, \dots, L$$

在 BiLSTM+CRF 中，这一个特征函数（发射概率）直接使用 LSTM 的输出计算得到，LSTM 可以计算出每一时刻位置对应不同标签的概率（如 '什' 对应 sbme 标签的概率分别为 0.8, 0.6, 0.4 和 0.4）

CRF 的第二个特征函数是状态转移特征函数，表示从一个状态  $y_1$  转移到另一个状态  $y_2$  的概率

$$\gamma_k t_k(y_{i-1}, y_i, x, i) \quad k = 1, \dots, K$$

CRF 的状态转移特征函数可以用一个状态转移矩阵表示，在训练时需要调整状态转移矩阵的元素值，前面分词任务中的 CRF 转移矩阵就可以表示为

	s	b	m	e
s	0.3	0.7	0	0
b	0	0	0.5	0.5
m	0	0	0.4	0.6
e	0.4	0.6	0	0

### 3.2 BiLSTM-CRF

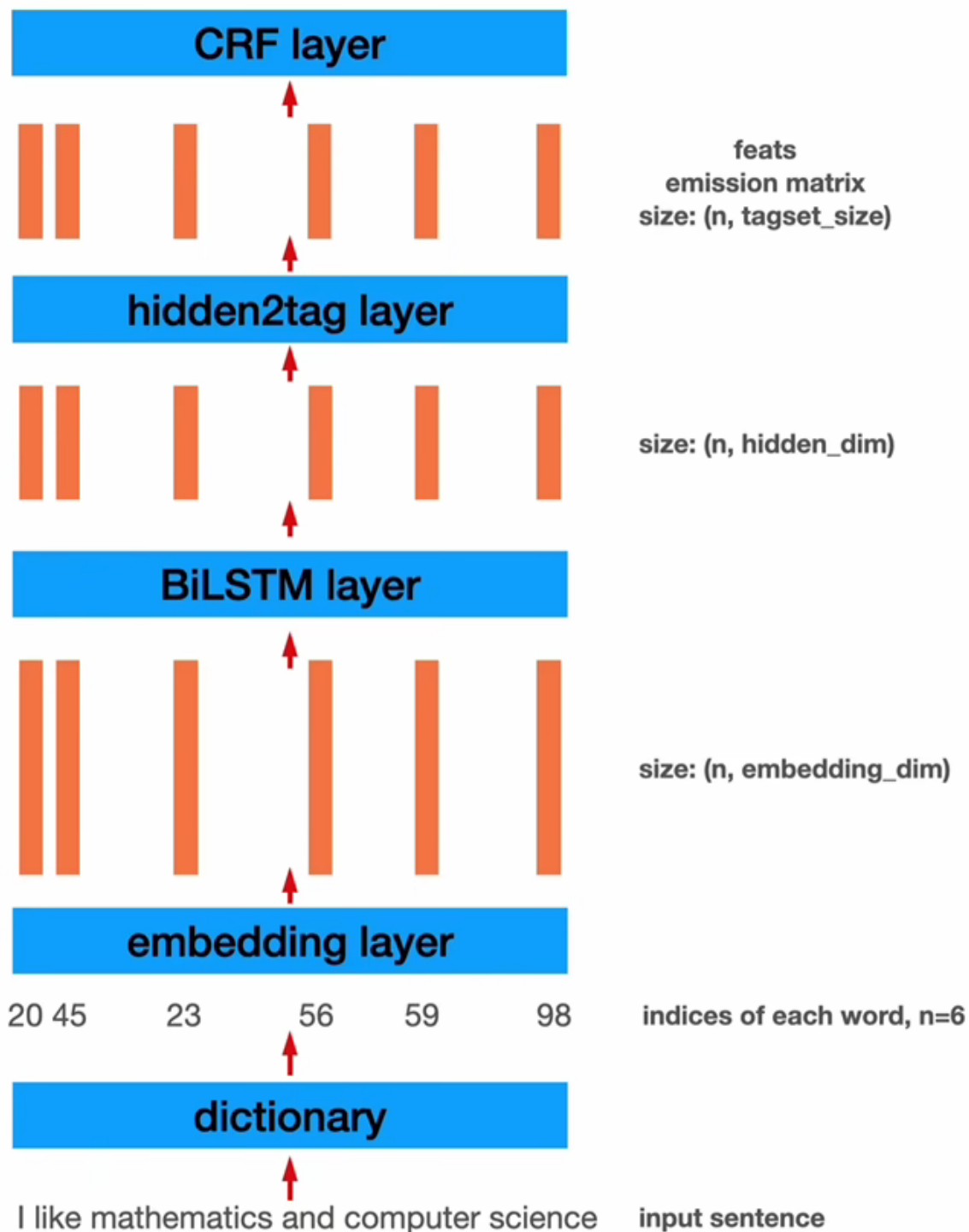
一个最基本的BiLSTM-CRF网络模型框架如下

```
class BiLSTM_CRF(nn.Module):
    def __init__(self, vocab_size, tag2idx, embedding_dim, hidden_dim):
        self.word_embeds = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim // 2,
                              num_layers=1, bidirectional=True)

        # 对应 CRF 的发射概率，即每一个位置对应不同类标的概率
        self.hidden2tag = nn.Linear(hidden_dim, self.tagset_size)

        # 转移矩阵，维度等于标签数量，表示从一个标签转移到另一标签的概率
        self.transitions = nn.Parameter(
            torch.randn(len(tag2idx), len(tag2idx)))
```

其大致结构可以表示为



给定一个句子 $x$ ，其标签序列为 $y$ 的概率使用如下公式计算

$$p(y|x) = \frac{\exp(\text{score}(x, y))}{Z(x)}$$

$$Z(x) = \sum_{y'} \exp(\text{score}(x, y'))$$

公式中的 $z(x)$ 表示所有标签序列打分的指数和，假如序列的长度为 $i$ ，标签的个数为 $k$ ，则序列的数量为 $k^i$ ，这无法直接计算，需要使用前向算法进行计算

公式中的score需要使用下面的式子计算，其中的Emit对应发射概率，即LSTM的输出概率，Trans对应了转移概率即CRF状态转移矩阵中对应的数值

$$score(x, y) = \sum_i Emit(x_i, y_i) + Trans(y_{i-1}, y_i)$$

BiLSTM-CRF采用最大似然法进行训练，其对应的损失函数如下

$$P(y | x) = \frac{\exp(score(x, y))}{\sum_{all\ possible\ \tilde{y}} \exp(score(x, \tilde{y}))}$$

$$-logP(y | x) = -log \frac{\exp(score(x, y))}{\sum_{all\ possible\ \tilde{y}} \exp(score(x, \tilde{y}))}$$

$$= -score(x, y) + log \sum_{all\ possible\ \tilde{y}} \exp(score(x, \tilde{y}))$$

损失函数

公式中的logZ(x)需要使用前向算法计算，这里不做介绍，详情参考[彻底了解 BiLSTM 和 CRF 算法-pytorch bilstm\\_crf \(51cto.com\)](#)，损失函数的计算；

### 3.3 viterbi算法

训练好模型后，预测过程需要用 viterbi 算法对序列进行解码，一些使用的符号意义如下

$i = 1, \dots, T$  表示序列长度

$j = 1, \dots, m$  表示标签种类

$\delta_i(j)$ 表示  $i$  时刻标签为  $j$  的最大概率

$\psi_i(j)$ 表示  $i$  时刻标签为  $j$  取得最大概率时前一时刻的标签

基于上述符号，viterbi算法的递推公式如下

$$\psi_0(j) = 0$$

$$\delta_0(j) = \begin{cases} 0 & j = Start \\ -1000 & j \neq Start \end{cases}$$

$$\delta_i(j) = \max_{1 \leq j' \leq m} \delta_{i-1}(j') + p_i(j) + T(j', j)$$

$$\psi_i(j) = \operatorname{argmax}_{1 \leq j' \leq m} \delta_{i-1}(j') + T(j', j)$$

基于上述递推式计算得到的 $\delta_i(j)$ 和 $\psi_i(j)$ 向前标注序列



则可以得到最后时刻最有可能的标签  $y_T^* = \operatorname{argmax}_{1 \leq j \leq m} \delta_T(j)$

根据 $\psi_i(j)$ 找回之前时刻的标签  $y_{i-1}^* = \psi_i(y_i^*)$

## 二、实体识别

### 1. 数据集介绍

原始语料库如下

 暴雨洪涝时间标签	2023/3/16 10:06	文件夹
 暴雨洪涝位置标签	2021/4/16 11:39	文件夹

实验任务是基于深度学习或预训练模型编程实现暴雨洪涝中文文本中的发生时间和发生地点两类实体的识别和抽取，这意味着需要将两类训练语料分别处理过后一起作为训练数据放入模型进行训练；

首先需要对原始语料进行处理，基于时间标签的文本内容如下

```
1 | "8·12"暴雨洪涝致甘肃8州市受灾 死亡43人
2 | 2018年08月19日 18:42
3 | 中国新闻网
4 | 中新网兰州8月19日电
5 | (记者
6 | 刘新梅)
7 | 甘肃省人民政府19日下午召开新闻发布会称，8月11日/05 至 12日/00
8 | ，甘肃陇南、天水、定西、甘南、兰州、庆阳、临夏、平凉等8个州市的942个县市区、271个乡镇、2241个村遭遇特大暴雨洪涝灾害，致159.38万人受灾，因灾死亡43人、失踪25人、受伤339人。目前已紧急转移安置15.9万人。
9 | 甘肃省民政厅副厅长徐亚荣说，据初步统计，"8·12"特大暴雨洪涝灾害造成8个州市的农作物受灾面积7.71万公顷，其中成灾5.99万公顷、绝收1.15万公顷，毁坏耕地0.66万公顷，倒塌房屋1.93万间、损坏房屋5.43万间，直接经济损失42.46亿元。目前，各地
10 | 灾区正在进一步核实灾情。完
```

基于位置标签的文本内容如下

```
1 | "8·12"暴雨洪涝致甘肃8州市受灾 死亡43人
2 | 2018年08月19日 18:42
3 | 中国新闻网
4 |
5 | 中新网兰州8月19日电(记者刘新梅) 甘肃省/LOC 政府新闻办19日下午召开新闻发布会称，8月11日至12日，甘肃/LOC 陇南/LOC 、天水/LOC 、定西/LOC 、甘南/LOC 、兰州/LOC 、庆阳/LOC 、临夏/LOC 、平凉/LOC
6 | 等8个州市的942个县市区、271个乡镇、2241个村遭遇特大暴雨洪涝灾害，致159.38万人受灾，因灾死亡43人、失踪25人、受伤339人。目前已紧急转移安置15.9万人。
7 | 甘肃省/LOC
8 | 民政厅副厅长徐亚荣说，据初步统计，"8·12"特大暴雨洪涝灾害造成8个州市的农作物受灾面积7.71万公顷，其中成灾5.99万公顷、绝收1.15万公顷，毁坏耕地0.66万公顷，倒塌房屋1.93万间、损坏房屋5.43万间，直接经济损失42.46亿元。目前，各地
9 | 灾区正在进一步核实灾情。完
```

观察可以发现，前几行对于训练来说毫无意义（因为标题、新闻发布时间以及新闻发布渠道对于文本时间和位置没有任何关系，同时这些内容也没有人工标注），为了减少网络的运算量也为了尽量保持数据集的整洁性，只需要获取正文内容即可，正文内容基本都是从第四行开始，因此对所有文本从第四行开始读取并将其融合为一行

```
#对data目录中的每个文本分别进行正文提取，并将其放入数据集中（时间和地点的训练数据可以放在一起，一行是一个新闻正文）
target_file=open('origin_data.txt','a',encoding='utf-8')
    #初始训练数据，注意是追加写，因为需要将时间和位置汇总，正确情况下一共有1929条数据
#提取data文件夹中所有文件的正文
datapath = 'D:\My_document\大三下文档\知识图谱\作业\第二次作业\实验2语料\暴雨洪涝时间标签'
#时间标签文本
# datapath = 'D:\My_document\大三下文档\知识图谱\作业\第二次作业\实验2语料\暴雨洪涝位置标签'
#位置标签文本
dirs = os.listdir(datapath)
    #dirs得到所有txt文件名
# for dir in dirs:
#     print(dir)
for dir in dirs:
    fname = datapath+'\\'+dir
```

```
# print(fname)
with open(fname,'r',encoding='utf-8')as src_file:
    lines = src_file.readlines()
    dirtyid = [0,1,2]
    str_new = ''
    # 提取新闻正文
    for x in range(len(lines)):
        if x not in dirtyid:
            str_new=str_new+lines[x].rstrip('\n')
    print(str_new)
    target_file.write(str_new.replace(' ','@')+'\n')
# 将空格替换成 '@', 为了之后方便检测实体边界
target_file.close()
```

处理过后的文本形式如下

记者2日从国家防汛抗旱总指挥部办公室获悉，今年以来全国因洪涝灾害死亡377人，失踪94人。据国家防办最新统计，今年以来的洪涝灾害共造成28省(区、市)5458万人受灾，因灾死亡377人，失踪94人，农作物受灾4581千公顷，倒塌房屋16万间。与2009年以来同期相比，洪涝灾害受灾人口偏少五成，死亡人口偏少七成，农作物受灾面积偏少五成，倒塌房屋偏少八成多。与去年同期相比，受灾人口、死亡人口、直接经济损失均偏少四成，农作物受灾面积均偏少五成，倒塌房屋偏少六成。刚刚过去的8月，全国共发生2次较强降水过程，降水量与常年基本持平但分布不均，有21省(区、市)590万人遭受洪涝灾害，其中@贵州/L0C@、@重庆/L0C@、@福建/L0C@、@浙江/L0C@受灾较重。(记者林晖)

这样处理过后的文本一共有1929行，为了之后方便处理（因为原始的时间标签和位置标签都有些问题，直接处理1929条数据会不断报错），将这些数据以200行一个文本的形式区分

```
# 将原始数据按照200行一组进行分组 -- 位置标签和时间标签都有问题
###将txt文件按行数分成多个txt文件
#open_diff = open('C:/Users/LitmoonHoney/Desktop/1.txt', 'r') # 源文本文件
#encoding=utf-8意思是编码格式为UTF-8格式，可用于多种语言的字符，包括中文。上一行这种表达，
txt文档有中文就报错
open_diff = open('original_data.txt', 'r',encoding='UTF-8') # 源文本文件
diff_line = open_diff.readlines()

line_list = []
for line in diff_line:
    line_list.append(line)

count = len(line_list) # 文件行数
n = 200 #按多少行来分割txt文件
print('源文件数据行数: ',count)
# 切分diff
diff_match_split = [line_list[i:i+n] for i in range(0,len(line_list),n)]# 每个文件的数据行数

# 将切分的写入多个txt中
for i,j in zip(range(0,int(count/n+1)),range(0,int(count/n+1))): # 写入txt，计算需要写入的文件数
    with open('D:/My_code/jupyter notebook/大三下作业/EntityExt/test_data/%d.txt'%(j+1),'w+',encoding='UTF-8') as temp:
        for line in diff_match_split[i]:
            temp.write(line)
print('拆分后文件的个数: ',i+1)
```

拆分后的文件形式如下

1.txt	2023/3/23 16:06	TXT 文件	271 KB
2.txt	2023/3/23 14:39	TXT 文件	279 KB
3.txt	2023/3/23 10:29	TXT 文件	267 KB
4.txt	2023/3/23 15:50	TXT 文件	255 KB
5.txt	2023/3/23 15:50	TXT 文件	165 KB
6.txt	2023/3/23 15:50	TXT 文件	271 KB
7.txt	2023/3/23 15:50	TXT 文件	242 KB
8.txt	2023/3/23 15:51	TXT 文件	257 KB
9.txt	2023/3/23 15:51	TXT 文件	245 KB
10.txt	2023/3/23 18:11	TXT 文件	241 KB

接下来就需要对每行的文本按照BEMO规则进行字标注，已知时间标签主要有DS D0 T0 TS，位置标签主要有LOC；

一开始想法很简单，直接按照规则 @地名/LOC@ 这种形式检测开始标志@和结束标志@即可提取出实体和对应标签，但实际上原始语料中还存在 @地名/LOC@地名/LOC@ 这种形式，因为没有很好的办法从前往后对实体进行界定，所以打算手动对这些不规范的实体进行调整，但是经过实际尝试发现人工标注数据太耗费时间了，因此初次进行数据处理的办法失效；

经过观察发现，文本中常见的几种标记方式主要有 @四川/LOC@、@四川/LOC@西安/LOC@、@四川、@/LOC@、中国/LOC@ 这几种形式，实际上能够提取的合法实体形式应该是 四川/LOC，既然从前往后界定实体边界失效，考虑从后往前（具体实现的时候就是将文本翻转），逐字符进行比对，开始标志为出现符号'@'且其后紧跟字符'C'，读取实体名称，以字符'@'或其他标点符号为结束标志（避免 中国/LOC@ 这种没有结束标志的实体）；

```
#将文本翻转过后进行提取并标注，最后再将文本翻转回来，需要注意LOC和时间的文本处理稍微有点区别
# input_data = codecs.open('D:/My_code/jupyter notebook/大三下作业/EntityExt/clear_data.txt','r','utf-8')
output_data = codecs.open('D:/My_code/jupyter notebook/大三下作业/EntityExt/complete_data/word2id_re_loc.txt','a',encoding='utf-8') #在测试的阶段可以不追加，但是实际阶段是追加在一起
datapath = 'D:/My_code/jupyter notebook/大三下作业/EntityExt/LOC_data'
dirs = os.listdir(datapath)

for dir in dirs:
    print('当前处理文件路径: '+datapath+'\\'+dir)
    input_data = codecs.open(datapath+'\\'+dir,'r','utf-8')
    for line in input_data.readlines():
        line=line.strip()
        text_re=line[::-1]
        i=0
        while i < len(text_re):
            #print(text_re[i])
            if text_re[i] == '@' and text_re[i+1] == 'C': # 这种判断会出现问题即@四川/LOC@作为开头那么最后text_re+1会直接溢出，因此需要判断当前长度
                # print('Find!')
                i=i+5 #跳过 @COL/
                content=''
```



```

        while text_re[i] not in punc:    # 标点符号作为结束的标志而非仅仅是@, 避免
出现 中国/LOC@这种
            content=content+text_re[i] # 最后结束的时候@不能+1跳过, 因为很可能是
@COL/@川四@COL/安西@形式交给条件判断即可, 之后会按照标点切分为短句所以没影响
            i=i+1
        #
        print(content)
        if len(content)!=0:            # 假如文本非空, 避免出现@/LOC@这种情况,
则进行字标注
            output_data.write(' '+COL_E/'+content[0])
            for j in content[1:len(content)-1]:
                output_data.write(' '+COL_M/'+j)
            output_data.write(' '+COL_B/'+content[-1])
        #
        if text_re[i] in punc_2:        # 假如是 @(内蒙古)@@克一河地区/LOC@
这种形式, 会在)停下, 此时应当直接跳过该标点符号
        #
            i=i+1
            if i==len(text_re)-1 and text_re[i]=='@':    # 假如此时@刚好是文章的开
头, 即i刚好循环到len()-1, 则直接结束本次循环
                i=i+1

            else:
                output_data.write(' O/'+text_re[i])
                i=i+1
                if i==len(text_re)-1 and text_re[i]=='@':    # 假如此时@刚好是文章的开
头, 即i刚好循环到len()-1, 则直接结束本次循环
                    output_data.write('\n')
                    break    # 假如此时是 @(内蒙
古)@@克一河地区/LOC@ 这种形式, 需要在else中对@进行跳过
                output_data.write('\n')
input_data.close()
output_data.close()

```

处理时间标签的代码形式和位置标签的代码类似, 这里给出不同的部分

```

# 下面是处理时间文本的操作
# 时间标签主要有 DS DO TO TS
# 行数是正确的, 只是不知道什么原因可能会多出一些空行, 之后会处理掉没关系
for dir in dirs:
    print('当前处理文件路径: '+datapath+'\\'+dir)
    input_data = codecs.open(datapath+'\\'+dir, 'r', 'utf-8')
    for line in input_data.readlines():
        line=line.strip()
        text_re=line[::-1]
        i=0
        while i < len(text_re):
            if text_re[i] == '@' and text_re[i+1] == 'S' and text_re[i+2] == 'D':
                i=i+4 #跳过 @SD/
                content=''

            while text_re[i] not in punc:
                content=content+text_re[i] # 最后结束的时候@不能+1跳过
                i=i+1
            if len(content)!=0:            # 假如文本非空, 避免出现@/LOC@这种情况,
则进行字标注

```







```
#接着将长句按照标点符号切分为短句
with open('D:/My_code/jupyter notebook/大三下作
业/EntityExt/complete_data/word2id_new.txt','rb') as inp:
    texts = inp.read().decode('utf-8')
    sentences = re.split('[, 。 ! ? 、 ‘ ’ “ ” ( ) ]/[0]'.encode('utf-8'),texts).decode('utf-8'),
    texts)
    output_data = codecs.open('D:/My_code/jupyter notebook/大三下作
业/EntityExt/complete_data/wordtagsplit.txt','w','utf-8')
    for sentence in sentences:
        if sentence != " ":
            output_data.write(sentence.strip()+'\n')
    output_data.close()
```

处理过后的文本形式如下

记/者/ 2/0 日/0 从/0 国/0 家/0 防/0 汛/0 抗/0 旱/0 总/0 指/0 挥/0 部/0 办/0 公/0 室/0 获/0 悉/0  
今/年/0 以/0 来/0 全/0 国/0 因/0 洪/0 涝/0 灾/0 害/0 死/0 亡/0 3/0 7/0 7/0 人/0  
失/踪/0 9/0 4/0 人/0  
据/0 国/0 家/0 防/0 办/0 最/0 新/0 统/0 计/0  
今/年/0 以/0 来/0 的/0 洪/0 涝/0 灾/0 害/0 共/0 造/0 成/0 2/0 8/0 省/0 (0 区/0  
市/0 )/0 5/0 4/0 5/0 8/0 万/0 人/0 受/0 灾/0  
因/0 灾/0 死/0 亡/0 3/0 7/0 7/0 人/0  
失/踪/0 9/0 4/0 人/0  
农/作/物/受/灾/0 4/0 5/0 8/0 1/0 千/0 公/0 顷/0  
倒/塌/0 房/0 屋/0 1/0 6/0 万/0 间/0  
与/0 2/0 0/0 0/0 0/0 年/0 以/0 来/0 同/0 期/0 相/0 比/0  
洪/涝/0 灾/0 害/0 受/0 灾/0 人/0 口/0 偏/0 少/0 五/0 成/0  
死/亡/0 人/0 口/0 偏/0 少/0 七/0 成/0  
农/作/物/受/灾/0 面/0 积/0 偏/0 少/0 五/0 成/0  
倒/塌/0 房/0 屋/0 偏/0 少/0 八/0 成/0 多/0  
与/0 去/0 年/0 同/0 期/0 相/0 比/0  
受/0 灾/0 人/0 口/0  
死/亡/0 人/0 口/0  
直/接/0 经/0 济/0 损/0 失/0 偏/0 少/0 四/0 成/0  
农/作/物/受/灾/0 面/0 积/0 均/0 偏/0 少/0 五/0 成/0  
倒/塌/0 房/0 屋/0 偏/0 少/0 六/0 成/0  
刚/刚/0 过/0 去/0 的/0 8/0 月/0  
全/0 国/0 共/0 发/0 生/0 2/0 次/0 较/0 强/0 降/0 水/0 过/0 程/0  
降/水/0 量/0 与/0 常/0 年/0 基/0 本/0 持/0 平/0 但/0 分/0 布/0 不/0 均/0  
有/0 2/0 1/0 省/0 (0 区/0  
市/0 )/0 5/0 9/0 0/0 万/0 人/0 遭/0 受/0 洪/0 涝/0 灾/0 害/0  
其/0 中/0 @/0 贵/B\_LOC 州/E\_LOC  
@/0 重/B\_LOC 庆/E\_LOC  
@/0 福/B\_LOC 建/E\_LOC  
@/0 浙/B\_LOC 江/E\_LOC 受/0 灾/0 较/0 重/0  
(0 记/者/0 林/0 晖/0 )/0

最后只需要将汉字对应的id（自定义）和上述已经标注好的规则标签对应存入句子数组中打包为pk1文件即可

```
# 分别将汉字对应的id和对应的tag转换为数组存入pk1文件中
def data2pk1():
    datas = list()
    labels = list()
    linedata=list()
    linelabel=list()
    tags = set()

    input_data = codecs.open('D:/My_code/jupyter notebook/大三下作
业/EntityExt/complete_data/wordtagsplit.txt','r','utf-8')
    for line in input_data.readlines():
        line = line.split()
        linedata=[]
        linelabel=[]
        numNot0=0
```

```

        for word in line:
            word = word.split('/')
            linedata.append(word[0])
            linelabel.append(word[1])
            tags.add(word[1])
            if word[1]!='0':
                numNot0+=1
        if numNot0!=0:
            datas.append(linedata)
            labels.append(linelabel)

input_data.close()
print(len(datas),tags)
print(len(labels))

#from compiler.ast import flatten
all_words = flatten(datas)
sr_allwords = pd.Series(all_words)
sr_allwords = sr_allwords.value_counts()
set_words = sr_allwords.index
set_ids = range(1, len(set_words)+1)

tags = [i for i in tags]
tag_ids = range(len(tags))
word2id = pd.Series(set_ids, index=set_words)
id2word = pd.Series(set_words, index=set_ids)
tag2id = pd.Series(tag_ids, index=tags)
id2tag = pd.Series(tags, index=tag_ids)

word2id["unknow"] = len(word2id)+1
print(word2id)
max_len = 60
def X_padding(words):
    ids = list(word2id[words])
    if len(ids) >= max_len:
        return ids[:max_len]
    ids.extend([0]*(max_len-len(ids)))
    return ids

def y_padding(tags):
    ids = list(tag2id[tags])
    if len(ids) >= max_len:
        return ids[:max_len]
    ids.extend([0]*(max_len-len(ids)))
    return ids

df_data = pd.DataFrame({'words': datas, 'tags': labels}, index=range(len(datas)))
df_data['x'] = df_data['words'].apply(X_padding)
df_data['y'] = df_data['tags'].apply(y_padding)
x = np.asarray(list(df_data['x'].values))
y = np.asarray(list(df_data['y'].values))

#调用sklearn的数据集划分函数，将原始数据集进行划分
from sklearn.model_selection import train_test_split

```

```
x_train,x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,
random_state=43)
x_train, x_valid, y_train, y_valid = train_test_split(x_train, y_train,
test_size=0.2, random_state=43)
```

调用pickle函数将最终结果写入 TheNews.pkl 文件中

```
import pickle
import os
with open('D:/My_code/jupyter notebook/大三下作业/EntityExt/complete_data/TheNews.pkl',
'wb') as outp:
    pickle.dump(word2id, outp)
    pickle.dump(id2word, outp)
    pickle.dump(tag2id, outp)
    pickle.dump(id2tag, outp)
    pickle.dump(x_train, outp)
    pickle.dump(y_train, outp)
    pickle.dump(x_test, outp)
    pickle.dump(y_test, outp)
    pickle.dump(x_valid, outp)
    pickle.dump(y_valid, outp)
print('** Finished saving the data.')
```

运行上述函数得到如下输出表示成功

```
11962 [' ', 'E_DO', 'M_LOC', 'M_DS', 'E_LOC', 'M_DO', 'B_DS', 'E_DS', 'E_TO', 'M_TO', 'B_DO', 'B_LOC', 'B_TO', 'E_TS', 'O', 'M_TS', 'B_TS']
11962
@      1
1      2
日      3
灾      4
2      5
...
碎      1834
船      1835
陡      1836
垵      1837
unknown 1838
Length: 1838, dtype: int64
** Finished saving the data.
```

在之后的训练过程中可以直接调用pickle的load方法读取 TheNews.pkl 文件中的数据，经过划分后的训练集和测试集数量分别是7655和2393；

## 2. 模块划分

### 2.1 模型介绍

这里主要介绍BiLSTM网络模型的搭建，也就是pycharm目录中的BiLSTMCRF.py文件；

BiLSTM-CRF网络主要分为embedding层、双向LSTM层以及全连接层，最终是单层CRF；

embedding层定义如下，如果使用了预训练词向量模型则调用vec.txt文件进行初始化

```
word_embeddings = tf.get_variable("word_embeddings",[self.embedding_size,
self.embedding_dim])
if self.pretrained:
    embeddings_init = word_embeddings.assign(self.embedding_pretrained)
input_embedded = tf.nn.embedding_lookup(word_embeddings, self.input_data)
input_embedded = tf.nn.dropout(input_embedded,self.dropout_keep)
```

接着是双向LSTM层，分为前向LSTM和反向LSTM，最终输出作为全连接层的输入

```
lstm_fw_cell = tf.nn.rnn_cell.LSTMCell(self.embedding_dim, forget_bias=1.0,
state_is_tuple=True)
lstm_bw_cell = tf.nn.rnn_cell.LSTMCell(self.embedding_dim, forget_bias=1.0,
state_is_tuple=True)
(output_fw, output_bw), states = tf.nn.bidirectional_dynamic_rnn(lstm_fw_cell,
                                                                    lstm_bw_cell,
                                                                    input_embedded,
                                                                    dtype=tf.float32,
                                                                    time_major=False,
                                                                    scope=None)
bilstm_out = tf.concat([output_fw, output_bw], axis=2)
```

全连接层对LSTM层的输出进行加权求和，最后使用tanh函数作为激活函数加入非线性因素

```
W = tf.get_variable(name="W", shape=[self.batch_size,2 * self.embedding_dim,
self.tag_size],
                    dtype=tf.float32)
b = tf.get_variable(name="b", shape=[self.batch_size, self.sen_len, self.tag_size],
                    dtype=tf.float32,
                    initializer=tf.zeros_initializer())
bilstm_out = tf.tanh(tf.matmul(bilstm_out, W) + b)
```

CRF层用于计算loss、转移矩阵以及likelihood

```
log_likelihood, self.transition_params = tf.contrib.crf.crf_log_likelihood(bilstm_out,
self.labels, tf.tile(np.array([self.sen_len]),np.array([self.batch_size])))
loss = tf.reduce_mean(-log_likelihood)
```

然后使用viterbi算法计算序列以及score

```
self.viterbi_sequence, viterbi_score =
tf.contrib.crf.crf_decode(bilstm_out,self.transition_params,tf.tile(np.array([self.sen
_len]),np.array([self.batch_size])))
```

最后一步选择Adam优化器优化损失函数

```
optimizer = tf.train.AdamOptimizer(self.lr)
self.train_op = optimizer.minimize(loss)
```

## 2.2 训练模块

训练模块位于utils.py文件，与常规模型训练类似，通过一定数量的epochs来最小化训练数据上的损失函数；

函数接受参数包括 model（表示序列标注模型）、sess（表示 TensorFlow 的 session 对象）、saver（表示 TensorFlow 中的 saver 对象，用于保存训练好的模型）、epochs（表示训练的轮数）、batch\_size（表示每一批数据的大小）、data\_train（表示训练数据）、data\_test（表示测试数据）、id2word（表示将 id 转化为 word 的字典）、id2tag（表示将 id 转化为 tag 的字典）；

```
batch_num = int(data_train.y.shape[0] / batch_size)
batch_num_test = int(data_test.y.shape[0] / batch_size)
```

上述代码表示计算了训练数据和测试数据中每一批数据的数量，并且存储在 batch\_num 和 batch\_num\_test 变量中

进入epoch循环之后每次循环中会对训练数据进行一次遍历。在每次循环中，会遍历训练数据中的每一批数据，并将其输入模型进行训练。训练时，会将当前批次的数据 x\_batch 和 y\_batch 分别作为模型的输入和标签，并将其通过 feed\_dict 提供给 TensorFlow 计算图中的相应节点进行计算

```
for batch in range(batch_num):
    x_batch, y_batch = data_train.next_batch(batch_size)
    feed_dict = {model.input_data:x_batch, model.labels:y_batch}
    pre,_ = sess.run([model.viterbi_sequence,model.train_op], feed_dict)
    acc = 0
    if batch%100==0:
        for i in range(len(y_batch)):
            for j in range(len(y_batch[0])):
                if y_batch[i][j]==pre[i][j]:
                    acc+=1
        print ("train acc:",float(acc)/(len(y_batch)*len(y_batch[0])))
```

每训练完一定数量的批次后，计算当前的训练精度，并将其输出。具体来说，会遍历当前批次中的每一个样本，并且将预测的标签 pre 与真实的标签 y\_batch 进行比较，计算准确率。如果当前批次的编号是 100 的倍数，就会将当前的准确率输出到控制台上；

每训练完一轮后，会将训练好的模型保存到磁盘上。具体来说，每隔 3 轮会将当前训练的模型保存到名为 modelX.ckpt 的文件中，其中 X 表示当前的轮数。

```
if epoch%3==0:
    saver.save(sess, path_name)
    print ("model has been saved")
```

## 2.3 测试模块

test模块位于utils.py文件中，主要用于输入测试数据，对模型进行测试的函数。具体来说，它的输入包括一个已经训练好的模型(model)、一个已经启动的会话(sess)、一个词到id的映射表(word2id)、一个id到标签的映射表(id2tag)以及一个批处理大小(batch\_size)。

```
model,sess,word2id,id2tag,batch_size
```

函数的实现中，首先要求用户输入要测试的文本，并且将其按照标点符号进行切割

```
text = input("Enter your input: ").decode('utf-8');
text = re.split(u'[ , . ! ? 、 ‘ ’ “ ” ( ) ]', text)
```

然后，将文本中的每一个句子转换为一个包含词id的序列。如果某个词不在词表(word2id)中，则用"unknow"对应的id来代替

```
text_id=[]
for sen in text:
    word_id=[]
    for word in sen:
        if word in word2id:
            word_id.append(word2id[word])
        else:
            word_id.append(word2id["unknow"])
    text_id.append(padding(word_id))
```

对于每一个句子，如果长度不足(max\_len)，则将其填充到相应的长度

```
zero_padding=[]
zero_padding.extend([0]*max_len)
text_id.extend([zero_padding]*(batch_size-len(text_id)))
```

接着，将所有的句子打包成一个批次，并将其输入到模型中，得到模型的预测结果，最后，将预测结果转换为对应的实体(entity)并进行输出

```
feed_dict = {model.input_data:text_id}
pre = sess.run([model.viterbi_sequence], feed_dict)
entity = get_entity(text,pre[0],id2tag)
print ('result:')
for i in entity:
    print(i)
```

### 3. 实验结果

---

调用

```
python main.py
```

开启训练，训练输出结果如下

```
train acc: 0.010416666666666666
train acc: 0.79375
train acc: 0.8916666666666667
./model/model0.ckpt
model has been saved
train acc: 0.9234375
train acc: 0.9432291666666667
train acc: 0.9375
./model/model1.ckpt
train acc: 0.9401041666666666
train acc: 0.9114583333333334
train acc: 0.9432291666666667
./model/model2.ckpt
train acc: 0.9390625
train acc: 0.9411458333333333
```



```
train acc: 0.9515625
./model/model3.ckpt
model has been saved
train acc: 0.928125
train acc: 0.9458333333333333
train acc: 0.9515625
./model/model4.ckpt
train acc: 0.94375
train acc: 0.9307291666666667
train acc: 0.946875
./model/model5.ckpt
train acc: 0.9348958333333334
train acc: 0.9411458333333333
train acc: 0.9614583333333333
./model/model6.ckpt
model has been saved
precision: 0.9846878680800942
recall: 0.27903871829105475
F1: 0.43485045513654097
train acc: 0.9572916666666667
train acc: 0.9666666666666667
train acc: 0.9661458333333334
./model/model7.ckpt
train acc: 0.965625
train acc: 0.9640625
train acc: 0.9708333333333333
./model/model8.ckpt
train acc: 0.9828125
train acc: 0.9671875
train acc: 0.98125
./model/model9.ckpt
model has been saved
precision: 0.8604553119730185
recall: 0.6812416555407209
F1: 0.7604321907600595
train acc: 0.9796875
train acc: 0.9786458333333333
train acc: 0.9864583333333333
./model/model10.ckpt
train acc: 0.978125
train acc: 0.9598958333333333
train acc: 0.9833333333333333
./model/model11.ckpt
train acc: 0.975
train acc: 0.978125
train acc: 0.9927083333333333
./model/model12.ckpt
model has been saved
precision: 0.9616963064295485
recall: 0.7039385847797063
F1: 0.8128733860088648
train acc: 0.9880208333333333
train acc: 0.9697916666666667
train acc: 0.9875
```

```
./model/model113.ckpt
train acc: 0.9744791666666667
train acc: 0.9848958333333333
train acc: 0.9755208333333333
./model/model114.ckpt
train acc: 0.9807291666666667
train acc: 0.9911458333333333
train acc: 0.9692708333333333
./model/model115.ckpt
model has been saved
precision: 0.889168765743073
recall: 0.7069425901201603
F1: 0.7876534027519525
train acc: 0.9723958333333333
train acc: 0.9890625
train acc: 0.9765625
./model/model116.ckpt
train acc: 0.9822916666666667
train acc: 0.9666666666666667
train acc: 0.9828125
./model/model117.ckpt
train acc: 0.9916666666666667
train acc: 0.9859375
train acc: 0.9895833333333334
./model/model118.ckpt
model has been saved
precision: 0.8612993224392188
recall: 0.7212950600801068
F1: 0.7851044504995459
train acc: 0.9817708333333334
train acc: 0.9729166666666667
train acc: 0.9791666666666666
./model/model119.ckpt
train acc: 0.978125
train acc: 0.9828125
train acc: 0.9932291666666667
./model/model120.ckpt
train acc: 0.9807291666666667
train acc: 0.9854166666666667
train acc: 0.9927083333333333
./model/model121.ckpt
model has been saved
precision: 0.8492553577915002
recall: 0.780373831775701
F1: 0.8133588450165247
train acc: 0.9859375
train acc: 0.9807291666666667
train acc: 0.9802083333333333
./model/model122.ckpt
train acc: 0.9828125
train acc: 0.9791666666666666
train acc: 0.9729166666666667
./model/model123.ckpt
train acc: 0.9869791666666666
```

```
train acc: 0.9671875
train acc: 0.990625
./model/model24.ckpt
model has been saved
precision: 0.8716715976331361
recall: 0.7867156208277704
F1: 0.8270175438596491
train acc: 0.9890625
train acc: 0.9947916666666666
train acc: 0.9869791666666666
./model/model25.ckpt
train acc: 0.9890625
train acc: 0.9848958333333333
train acc: 0.9901041666666667
./model/model26.ckpt
train acc: 0.9869791666666666
train acc: 0.9911458333333333
train acc: 0.9942708333333333
./model/model27.ckpt
model has been saved
precision: 0.8743248109470652
recall: 0.8104138851802403
F1: 0.841157110687684
train acc: 0.9880208333333333
train acc: 0.9916666666666667
train acc: 0.990625
./model/model28.ckpt
train acc: 0.9807291666666667
train acc: 0.9817708333333334
train acc: 0.9828125
./model/model29.ckpt
train acc: 0.9921875
train acc: 0.9885416666666667
train acc: 1.0
./model/model30.ckpt
model has been saved
precision: 0.8790931989924433
recall: 0.8154205607476636
F1: 0.8460606060606061
```

经过模型训练得到上述实验结果，主要评价指标为查全率、查准率以及F1分数：

- 查全率 (recall)：模型在测试集中能够正确预测出的正样本数量与测试集中所有正样本数量的比例为 X%。
- 查准率 (precision)：模型在测试集中预测为正样本的样本中，表示真正是正样本的数量与所有被预测为正样本的样本数量的比例为 Y%。
- F1分数 (F1 score)：F1分数综合了查准率和查全率的指标，其计算公式为  $2 * (\text{precision} * \text{recall}) / (\text{precision} + \text{recall})$ ;

其中，查全率衡量了模型能够识别出所有的正样本的能力，而查准率则衡量了模型在预测正样本时的准确性。F1分数是综合了这两个指标，用于综合评估模型的性能。在我们的实验中，我们得到了一个具有较高查全率和查准率的模型，最终其F1分数0.84，表明该模型在预测正样本时具有很好的准确性和召回率。

观察实验结果可知BiLSTMCRF模型具备很快的收敛速度，但是也可以观察到一方面是模型的学习能力过强，一方面因为模型的训练数据不足，在训练过程中出现了过拟合的现象，最终对训练得到的模型进行测试，输入命令

```
python main.py test
```

前面说过一方面模型学习能力太强，一方面我们的数据因为是自己清洗的所以可能并不是特别标准，同时数据量不够，这些原因都可能导致模型的泛化能力变弱，具体表现就是我们在测试的过程中会出现如下情况

```
Enter your input: 据应急管理部统计，7月份以来洪涝灾害造成 江西、 安徽 、 湖北、 湖南、 重庆 、 贵州 等24省(区、市)2027.2  
万人次受灾  
result:  
_LOC:湖南  
Enter your input: 据统计，7月份以来洪涝灾害造成 江西、 安徽、 湖北 、 湖南、 重庆、 贵州 等24省(区、市)2027.2万人次受灾  
result:  
_LOC:湖南
```

因此如果想要提高模型的泛化能力可以从以下几个方面入手（需要根据具体情况采取合适的方法才可能得到更好的结果）：

1. 更多的数据：收集更多的数据可以使模型接触到更多的变化和情况，提高模型的泛化能力。可以通过增加训练集数据、进行数据增强等方式来实现。
2. 正则化：正则化可以减少模型的复杂度，降低过拟合的风险，进而提高泛化能力。可以使用L1、L2正则化、dropout等方法。
3. 交叉验证：使用交叉验证可以评估模型的泛化能力，并帮助我们调整模型的参数，进而提高泛化能力。
4. 模型结构优化：可以通过增加或减少网络层数、调整每一层的神经元数、使用更优的激活函数等方法来调整模型结构，以提高泛化能力。
5. 集成学习：通过将多个模型进行集成，可以得到更好的结果，同时降低过拟合的风险，提高泛化能力。可以使用投票、堆叠等集成方法。
6. 对抗训练：对抗训练可以通过引入对抗样本，使得模型能够更好地抵御噪声、干扰等攻击，提高泛化能力。
7. 知识蒸馏：知识蒸馏可以通过将一个复杂的模型的知识迁移到一个简单的模型中，使得简单的模型也能够具有类似复杂模型的泛化能力。

## 4. 实验总结

本次实验要求基于给定的暴雨洪涝中文语料库，利用已人工标注的样本作为训练集和测试集，基于深度学习和预训练模型，编程实现暴雨洪涝中文文本中的发生时间和发生地点两类实体的识别和抽取。在完成实验的过程中我个人认为最难的一步在于数据的处理，吸收了上一次的教训后这次在数据处理的时间花费上不是很长，得到了在训练中能够得到较好结果的训练数据，但是本次实验中使用的数据集比较小，未来可以尝试更大的数据集进行实验。此外，除了BiLSTM-CRF还可以尝试其他的实体抽取模型，如基于注意力机制的模型。

