

**课 程 实 验 报 告**

**课程名称： 自然语言处理实验**

**专业班级： 计算机科学与技术1903**

**学 号： U201914995**

**姓 名： 罗虞阳**

**指导教师： 姚德中**

**报告日期： 2022年 6月 19日**

**计算机科学与技术学院**

目 录

[中文分词实现 2](#_Toc106740068)

[1 实验目的和问题描述 2](#_Toc106740069)

[1.1 实验目的 2](#_Toc106740070)

[1.2 基础任务 2](#_Toc106740071)

[1.3 选做任务 3](#_Toc106740072)

[2 模块设计-基于词典 3](#_Toc106740073)

[2.1 前向最大匹配 3](#_Toc106740074)

[2.2 后向最大匹配 3](#_Toc106740075)

[2.3 双向最大匹配 4](#_Toc106740076)

[3 模块设计-基于统计学习 4](#_Toc106740077)

[3.1 模块整体设计 5](#_Toc106740078)

[3.2 模块实现 6](#_Toc106740079)

[4 运行结果 7](#_Toc106740080)

[5 实验小结 8](#_Toc106740081)

[参考文献 10](#_Toc106740082)

[附录A 中文分词实现的源程序 11](#_Toc106740083)

[Dictionaru\_based.py 11](#_Toc106740084)

[data\_u.py 16](#_Toc106740085)

[dataloader.py 19](#_Toc106740086)

[infer.py 21](#_Toc106740087)

[mergy.py 23](#_Toc106740088)

[Model.py 26](#_Toc106740089)

[run.py 28](#_Toc106740090)

# 中文分词实现

## 1 实验目的和问题描述

中文分词指的是将一个汉字序列切分成一个一个单独的词。中文分词是文本挖掘的基础，对于输入的一段中文，成功的进行中文分词，可以达到电脑自动识别语句含义的效果。它是信息提取、信息检索、机器翻译、文本分类、自动文摘、语音识别、文本语音转换、自然语言理解等中文信息处理领域的基础。由于中文分词缺乏形式上的分解符，导致我们对词的认定会出现较大的偏差，这就体现了中文分词问题的困难性之一，即分词标准问题。

### 实验目的

通过实验了解中文分词的大致思路。在实验中实现中文分词处理，可考虑使用课堂讲解过的算法（比如基于统计、基于词典的分词方法等），或者课外学习算法（比如Bi-LSTM+CRF模型等）进行分词，最后对比不同算法分词效果和性能，加深对中文分词算法的理解。

### 1.2 基础任务

1. 实现基于词典的分词算法

实验一资料包下的“Dictionary\_based”文件夹中提供了基础词典和分词算法的大致框架。分词算法的核心部分需要大家完成，实验中提供了若干测试样本用以帮助大家判断算法是否正确实现。

1. 实现基于统计学习的分词算法

实验中给出Bi-LSTM+CRF模型的基础实现，相关代码及说明文档位于实验一资料包下的“Bi-LSTM+CRF”文件夹下。请根据给定的实验资料中README.md文件配置相应实验环境，说明：（1）提供源码PyTorch语言编写（可根据个人掌情况用其他语言编写），默认运行版本是CPU版本；（2）如希望运行NPU版本，大家可跟任课老师联系，申请华为云资源运行（需提前统计名单：姓名+学号+个人手机号码+邮箱）；

### 1.3 选做任务

优化基础任务中实现的分词器，可考虑的优化方案有：

1. 修改网络结构，例如引入BERT等预训练语言模型；
2. 与命名实体识别算法相互配合，减少对命名实体的错误分割；
3. 构造合适的词典集（可扩充+人工整理）；
4. 实现新词发现（登录）功能，识别测试集中的新词（未登录词）；
5. 调整、优化模型训练过程中的超参数。

完成优化后对测试文件“Bi-LSTM+CRF/data/test.txt”进行分词，分词结果保存到.txt文件中utf-8编码，词与词之间以空格分隔，每个测试样本占一行。文件“Bi-LSTM+CRF/cws\_result.txt”中给出了输出示例。提交分词结果后，依据单词级别的F1-score进行评判，决定选做部分的实验分数。

## 2 模块设计-基于词典

### 2.1 前向最大匹配

从待分词句子的左边向右边搜索，寻找词的最大匹配。我们需要规定一个词的最大长度，每次扫描的时候寻找当前开始长度为最大长度的词来和字典中的词匹配，如果没有找到，就缩短长度继续寻找，直到找到字典中的词或者成为单字。

算法流程如下：

（1）从待分词子串中从前往后取出max\_len个字，然后扫描分词字典，测试该max\_len个字的子串是否在字典中；

（2）如果存在，则从待分词子串中删除掉该max\_len个字的子串，重新按照规则取子串，重复（1）；

（3）如果不存在于字典中，则减少该子串的最右一个字，之后重复（1）。

### 2.2 后向最大匹配

从待分词句子的右边向左边搜索，寻找词的最大匹配。我们需要规定一个词的最大长度，每次扫描的时候寻找当前开始向左寻找长度为最大长度的词来和字典中的词匹配，如果没有找到，就缩短长度继续寻找，直到找到字典中的词或者成为单字。

算法流程如下：

（1）从待分词子串中从后往前取出max\_len个字，然后扫描分词字典，测试该max\_len个字的子串是否在字典中；

（2）如果存在，则从待分词子串中删除掉该max\_len个字的子串，重新按照规则取子串，重复（1）；

（3）如果不存在于字典中，则减少该子串的最左一个字，之后重复（1）。

### 2.3 双向最大匹配

将前向最大匹配算法和后向最大匹配算法进行比较，从而确定正确的分词方法，选取更为合适的作为结果。

双向最大匹配的算法流程如下：

（1）对句子分别做前向最大匹配和后向最大匹配，得到两种匹配结果；

（2）比较前向最大匹配和后向最大匹配结果，如果分词结果相同，返回其中任意一个；如果不同，则做（3）；

（3）如果分词结果不同：

1、比较两者分词总数量，数量高者罚分；

2、比较两者分词中单字词数量，单字词多者罚分；

3、比较两者分词中非字典词数量，非字典词多者罚分；

4、选择罚分最少的作为最终结果。

## 3 模块设计-基于统计学习

基于统计学习的系统主要分为数据预处理，模型构建和结果分析优化三个部分，系统各部分组成和结构如图3.1所示，其中每个模块的具体设计会在下一小节给出。

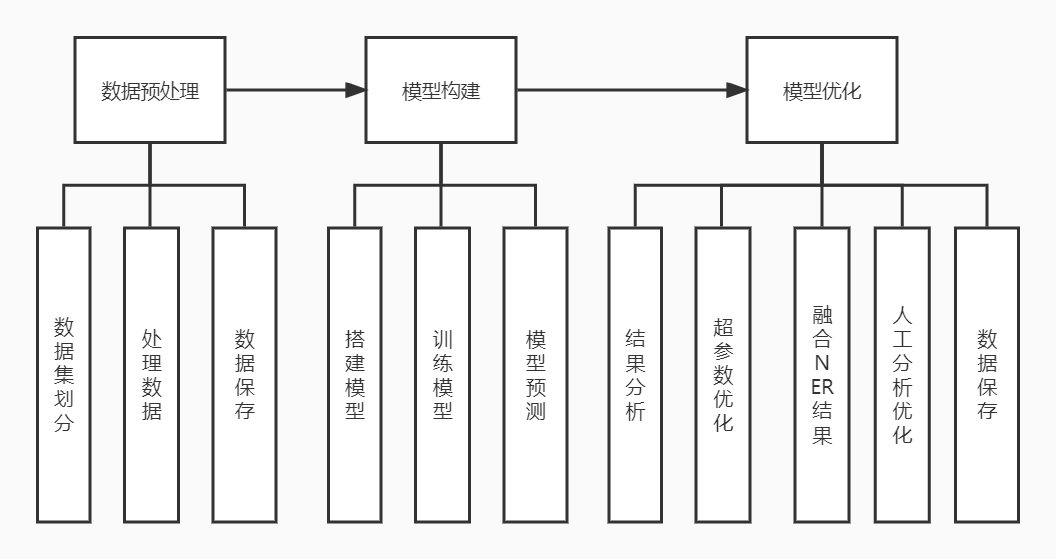


图3.1 系统组成和结构图

### 3.1 模块整体设计

中文分词模型主要分为数据预处理，模型构建和结果分析优化这三个部分，接下来介绍每个模块的详细设计和模块之间的关系。

首先是数据预处理模块。这个模块主要负责原始数据的读取，对原始数据进行编码，然后将编码结果打包保存到对应文件中。数据读取部分将训练数据读取，为每个字生成对应的BMES标签。基础模块编码部分将句子中的每个字替换成数据预处理部分生成的字典中的对应编码。在改进模块中，由于需要和bert模型输入输出相匹配，所以在每个句子的开头和结尾处添加了【cls】和【sep】两个标志，如果句子中的字未在字典中出现，则使用【unk】进行标注。由于每个句子的长度不相同，需要使用padding将每个句子扩充成相同长度，使用【pad】进行填充，但是由于这会引入许多冗余信息，所以还需要使用input mask和output mask进行掩码操作，相当于告诉模型什么位置的信息是有用的。最后在打包部分，将生成好的每个句子的编码结果保存到文件中，方便之后训练直接读取，避免了重复操作。

然后是模型构建模块。这一部分分为基础的BiLSTM+CRF和优化添加的bert部分。基础部分已经由老师给出，bert预训练模型使用的是RoBERTa-wwm-ext，该结构有哈工大讯飞实验室提供。为了获取更加丰富的编码信息，我将原始编码层替换成了预训练模型，这样能够通过attention机制获得语义信息更加丰富的句子编码，同时需要对参数embedding\_dim和hidden\_dim进行修改，以适配bert模型的输出。

最后是结果分析及优化模块。每次训练完一个epoch都会将相关信息，比如loss，准确率，f1 score等，记录到log.txt文件中。通过查看log文件中的信息，即可知道模型当前训练得如何。最后还将中文命名实体NER模型中输出的BIOES标注和中文分词BMES标注按照粗粒度优先规则进行融合，得到最后的分词标签。

### 3.2 模块实现

模型在实验提供的框架之上引入了bert预训练模型作为编码层，对数据预处理部分进行了修改以适配编码层，同时对模型的结果进行了简单的分析处理。

data\_u.py

1. get\_list：输入为分词，根据单词的长度生成对应的标注，然后查询tag2id将标注转化成编码进行输出。

(2) handle\_data：该函数负责处理所有训练数据。首先分词形成token list，使用get\_list函数对形成的token进行编码形成label list，然后对token list和label list进行裁剪，删除超出固定长度的部分，使用bert预训练模型的tokenizer求出input\_id,之后加入特殊符号【cls】和【sep】标注开始和结尾，然后对不满足固定长度的编号列表，标签列表和掩码进行填充，同时生成一个输出掩码用于屏蔽掉填充部分的标签。最后将处理好的数据保存到.pkl文件中，方便后续读取时避免重复操作。

model.py

1. \_\_init\_\_：整体在老师发的基础部分上进行了修改，使用了bert预训练好的tokenizer和word\_embeds做初始化。
2. Init\_hidden：使用基础代码。
3. \_get\_lstm\_features：将源代码中的编码层替换成了bert预训练模型，通过attention机制可以获得语义信息更加丰富的编码，经过lstm后获得特征并返回。
4. Forward：输入为input\_ids和input\_mask，使用\_get\_lstm\_features函数获得emissions，输入crf层获得loss值。
5. Infer：输入为input\_ids和input\_mask，使用\_get\_lstm\_features函数获得emissions，经过crf层解码后得到推断出的标签。

run.py

1. get\_param：获取参数。在源码上进行了少数修改，将embedding维数改为768适配bert的维数，其他的默认参数均在超参数优化进行了调整。
2. set\_logger：将log记录保存到log.txt中，这部分使用的是基础原始代码。
3. entity\_split：使用的基础原始代码。
4. main：首先获取处理好的数据，然后初始化模型，并对输入进行打乱操作。然后将数据集划分为训练集和测试集，比例为9:1。准备工作完成后，开始迭代训练。采用小批量梯度下降法，对模型进行训练，首先计算输入的每个batch的loss，然后backward()进行反向传播修改模型中权重的参数，并将记录信息保存。在训练集上训练完后，使用测试集进行测试，保存结果信息到log.txt中。

infer.py：

获取数据，并对数据进行处理，包括编码，截取和填充操作等，将处理好的数据和相应的掩码输入到模型中得到预测的结果，并放入文件中保存。

merge.py

读取分词结果，分词标签结果和中文命名实体识别结果的标签序列，将三者进行融合，采取的策略是粗粒度有限的合并策略，除非一句话在融合后句子中的分词数量减少，否则不改变分词结果。由于部分预测结果错误会导致中文命名实体的标注在B后缺少E，这里使用人工纠错的方式进行修正，以保证能够让merge.py程序能够顺利运行。

## 4 运行结果

利用实验包已给出的代码框架，实现完对应的匹配函数后，运行测试样例。

1. 基于词典的中文分词，得到输出结果如下：

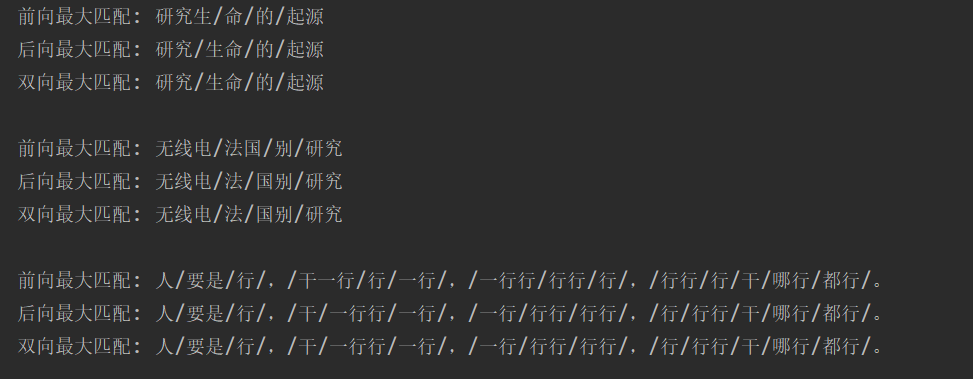


图 1-1 中文分词测试结果

1. 基于统计学习的中文分词，根据模型得到的三个结果，分别是原始模型结果，原始模型+中文命名实体结果和融合了bert预训练模型的结果，命名依次为result.txt, result\_cws\_ner.txt和result\_cwsbert\_ner.txt，保存在result文件夹中。

## 5 实验小结

实验一需要完成基于词典的分词算法，包括前向最大匹配，后向最大匹配和双向最大匹配。经过网上查阅资料和理解算法内容后，我完成了三中算法的编写，结果与测试样例的相同。

实验二是需要完成基于统计的分词算法，老师已经给了基础的代码实现，需要在基础模型之上进行优化，包括调整超参数，添加预训练模型和使用中文命名实体结果进行融合。由于在实验指导书中的在线测试网址无法访问，我联系了另一个班上的同学，使用他们提供的临时在线检测网站进行了测试。在这三种方法中，效果最好的是添加bert预训练模型，调整超参数是为了匹配bert预训练模型的编码层，而添加融合中文命名实体结果对结果的优化提升程度不是很大。同时预训练模型除了此次试验中使用的RoBERTa之外，还有ALBERT，ERNIE，XLNet，GPT-2等，由于光是训练RoBERTa就花费了将近三天的时间，由于时间不足和算力的缺乏，其他预训练模型的效果就没有测试和对比，未来有时间可以更换不同的与训练模型，比较效果。

通过这两个实验，我学习了解了经典的分词算法，对模型内部的具体实现有了基础的了解和动手尝试，尽管修改错误和查阅资料以及等待模型训练耗费了我大量的时间和尽力，往往是在图书馆从早上开始调试，然后训练等待结果，尤其是加入了bert预训练模型后训练时间更是达到了三天，但是最后结果有所提升仍是让我欣喜，也让我对未来的科研实验有了初步的了解。

本次实验进行了一个完整的针对某个NLP任务的数据分析处理，模型搭建，调参，模型优化的过程，是关于NLP相关实验的一次宝贵体验，我在这个过程中收获了很多相关专业知识，也对这个领域有了更深的了解。

# 参考文献

[1] 郑捷著. NLP汉语自然语言处理---原理与实践. 电子工业出版社

# 附录A 中文分词实现的源程序

## Dictionaru\_based.py

class Tokenizer(object):

def \_\_init\_\_(self, init\_words, init\_max\_len, init\_delimiter):

self.words = init\_words

self.max\_len = init\_max\_len

self.delimiter = init\_delimiter

def forward\_mm\_split(self, fmm\_text):

"""

正向最大匹配分词算法

:param fmm\_text: 待分词字符串

:return: 分词结果，以list形式存放，每个元素为分出的词

"""

# 字词列表，存放分词结果

word\_list = []

# 用于记录分词的起始位置

count = 0

# 字或词当前的长度

word\_len = self.max\_len

while word\_len > 0 and count < len(fmm\_text):

word = fmm\_text[count:count + word\_len]

word\_len = len(word)

if (word in self.words) or (word in self.delimiter):

word\_list.append(word)

count = count + word\_len

word\_len = self.max\_len

else:

word\_len = word\_len - 1

return word\_list

def reverse\_mm\_split(self, rmm\_text):

"""

逆向最大匹配分词算法

:param rmm\_text: 待分词字符串

:return: 分词结果，以list形式存放，每个元素为分出的词

"""

# 字词列表，存放分词结果

word\_list = []

# 用于记录分词的末尾位置

count = len(rmm\_text)

# 字或词当前的长度

word\_len = self.max\_len

while word\_len > 0 and count > 0:

if count <= word\_len:

word = rmm\_text[:count]

else:

word = rmm\_text[(count - word\_len):count]

word\_len = len(word)

if (word in self.words) or (word in self.delimiter):

word\_list.insert(0, word)

count = count - word\_len

word\_len = self.max\_len

else:

word\_len = word\_len - 1

return word\_list

def bidirectional\_mm\_split(self, bi\_text):

"""

双向最大匹配分词算法

:param bi\_text: 待分词字符串

:return: 分词结果，以list形式存放，每个元素为分出的词

"""

# 前向最大匹配得到的分词结果

forward = self.forward\_mm\_split(bi\_text)

# 后向最大匹配得到的分词结果

reverse = self.reverse\_mm\_split(bi\_text)

# 总词数

forward\_total\_words = len(forward)

reverse\_total\_words = len(reverse)

# 单字词个数

forward\_single\_words = 0

reverse\_single\_words = 0

# 非字典词数

forward\_illegal\_words = 0

reverse\_illegal\_words = 0

# 罚分，分值越低，表明结果越好

forward\_score = 0

reverse\_score = 0

if forward == reverse:

return forward

else:

# 统计前向匹配的各个词情况

for word in forward:

if len(word) == 1:

forward\_single\_words += 1

if word not in self.words:

forward\_illegal\_words += 1

# 统计后向匹配的各个词情况

for word in reverse:

if len(word) == 1:

reverse\_single\_words += 1

if word not in self.words:

reverse\_illegal\_words += 1

# 计算罚分

if forward\_total\_words < reverse\_total\_words:

reverse\_score += 1

else:

forward\_score += 1

if forward\_illegal\_words < reverse\_illegal\_words:

reverse\_score += 1

else:

forward\_score += 1

if forward\_single\_words < reverse\_single\_words:

reverse\_score += 1

else:

forward\_score += 1

# 比较罚分情况，罚分最小的选做最终结果

if forward\_score < reverse\_score:

return forward

else:

return reverse

def load\_dict(path):

tmp = set()

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

for line in f:

word = line.strip().split(' ')[0]

tmp.add(word)

return tmp

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

words = load\_dict('dict.txt')

max\_len = max(map(len, [word for word in words]))

delimiter = ',./;\'<>?:\"-=\_+!@#$%^&\*(){}，。？、；‘“’”·~[]'

# test

tokenizer = Tokenizer(words, max\_len, delimiter)

texts = [

'研究生命的起源',

'无线电法国别研究',

'人要是行，干一行行一行，一行行行行行，行行行干哪行都行。'

]

for text in texts:

# 前向最大匹配

print('前向最大匹配:', '/'.join(tokenizer.forward\_mm\_split(text)))

# 后向最大匹配

print('后向最大匹配:', '/'.join(tokenizer.reverse\_mm\_split(text)))

# 双向最大匹配

print('双向最大匹配:', '/'.join(tokenizer.bidirectional\_mm\_split(text)))

print('')

## data\_u.py

import codecs

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

import pickle

import torch

from transformers import AlbertModel, BertTokenizer

INPUT\_DATA = "train.txt"

SAVE\_PATH = "./datasave.pkl"

id2tag = ['B', 'M', 'E', 'S'] # B：分词头部 M：分词词中 E：分词词尾 S：独立成词

tag2id = {'B': 0, 'M': 1, 'E': 2, 'S': 3}

word2id = {}

id2word = []

tokenizer = BertTokenizer.from\_pretrained("./chinese-roberta-wwm-ext")

def get\_list(input\_str):

"""

单个分词转换为tag序列

:param input\_str: 单个分词

:return: tag序列

"""

output\_str = []

if len(input\_str) == 1:

output\_str.append(tag2id['S'])

elif len(input\_str) == 2:

output\_str = [tag2id['B'], tag2id['E']]

else:

M\_num = len(input\_str) - 2

M\_list = [tag2id['M']] \* M\_num

output\_str.append(tag2id['B'])

output\_str.extend(M\_list)

output\_str.append(tag2id['E'])

return output\_str

def handle\_data():

"""

处理数据，并保存至savepath

:return:

"""

input\_ids\_l = []

input\_mask\_l = []

label\_l = []

output\_mask\_l = []

line\_num = 0

with open(INPUT\_DATA, 'r', encoding="utf-8") as ifp:

for line in ifp:

line\_num = line\_num + 1

line = line.strip()

if not line:

continue

words = line.split()

sent = ''.join(words)

tokens = [i for i in sent]

label = []

for item in words:

label.extend(get\_list(item))

if len(tokens) > 512 - 2:

tokens = tokens[: (512 - 2)]

label = label[: (512 - 2)]

tokens\_cs = '[CLS]' + ' '.join(tokens) + '[SEP]'

tokenized\_text = tokenizer.tokenize(tokens\_cs)

input\_ids = tokenizer.convert\_tokens\_to\_ids(tokenized\_text)

input\_mask = [1] \* len(input\_ids)

label = [3] + label + [3]

while len(input\_ids) < 512:

input\_ids.append(0)

input\_mask.append(0)

while len(label) < 512:

label.append(-1)

output\_mask = [1] \* len(tokens)

output\_mask = [1] + output\_mask + [1]

while len(output\_mask) < 512:

output\_mask.append(0)

assert len(input\_ids) == 512

assert len(input\_mask) == 512

assert len(label) == 512

assert len(output\_mask) == 512

input\_ids\_l.append(input\_ids)

input\_mask\_l.append(input\_mask)

label\_l.append(label)

output\_mask\_l.append(output\_mask)

print(tokenizer.convert\_ids\_to\_tokens(input\_ids\_l[0]))

print(input\_ids\_l[0])

print(input\_mask\_l[0])

print(label\_l[0])

print(output\_mask\_l[0])

with open(SAVE\_PATH, 'wb') as outp:

pickle.dump(tag2id, outp)

pickle.dump(id2tag, outp)

pickle.dump(input\_ids\_l, outp)

pickle.dump(label\_l, outp)

pickle.dump(input\_mask\_l, outp)

pickle.dump(output\_mask\_l, outp)

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

handle\_data()

## dataloader.py

import torch

import pickle

from torch.utils.data import Dataset, DataLoader

from torch.nn.utils.rnn import pad\_sequence

class Sentence(Dataset):

def \_\_init\_\_(self, x, y, batch\_size=10):

self.x = x

self.y = y

self.batch\_size = batch\_size

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

return len(self.x)

def \_\_getitem\_\_(self, idx):

assert len(self.x[idx]) == len(self.y[idx])

return self.x[idx], self.y[idx]

@staticmethod

def collate\_fn(train\_data):

train\_data.sort(key=lambda data: len(data[0]), reverse=True)

data\_length = [len(data[0]) for data in train\_data]

data\_x = [torch.LongTensor(data[0]) for data in train\_data]

data\_y = [torch.LongTensor(data[1]) for data in train\_data]

mask = [torch.ones(l, dtype=torch.uint8) for l in data\_length]

data\_x = pad\_sequence(data\_x, batch\_first=True, padding\_value=0)

data\_y = pad\_sequence(data\_y, batch\_first=True, padding\_value=0)

mask = pad\_sequence(mask, batch\_first=True, padding\_value=0)

return data\_x, data\_y, mask, data\_length

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

# test

with open('./data/datasave.pkl', 'rb') as inp:

tag2id = pickle.load(inp)

id2tag = pickle.load(inp)

x\_train = pickle.load(inp)

y\_train = pickle.load(inp)

x\_test = pickle.load(inp)

y\_test = pickle.load(inp)

train\_dataloader = DataLoader(Sentence(x\_train, y\_train), batch\_size=10, shuffle=True, collate\_fn=Sentence.collate\_fn)

for input, label, mask, length in train\_dataloader:

print(input, label)

break

## infer.py

import torch

import pickle

from transformers import AlbertModel, BertTokenizer

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

model = torch.load('save/model\_epoch15.pkl', map\_location=torch.device('cpu'))

output = open('cws\_result.txt', 'w', encoding='utf-8')

label\_output = open('./data/cws\_res.txt', 'w', encoding='utf8')

tokenizer = BertTokenizer.from\_pretrained("voidful/albert\_chinese\_base")

with open('data/datasave.pkl', 'rb') as inp:

tag2id = pickle.load(inp)

id2tag = pickle.load(inp)

input\_ids\_l = pickle.load(inp)

label\_l = pickle.load(inp)

input\_mask\_l = pickle.load(inp)

output\_mask\_l = pickle.load(inp)

with open('data/test\_data.txt', 'r', encoding='utf-8') as f:

for test in f:

flag = False

test = test.strip()

tokens = [i for i in test]

tokens\_cs = '[CLS]' + ' '.join(tokens) + '[SEP]'

tokenized\_text = tokenizer.tokenize(tokens\_cs)

input\_ids = tokenizer.convert\_tokens\_to\_ids(tokenized\_text)

input\_mask = [1] \* len(input\_ids)

while len(input\_ids) < 512:

input\_ids.append(0)

input\_mask.append(0)

output\_mask = [1] \* len(tokens)

output\_mask = [1] + output\_mask + [1]

while len(output\_mask) < 512:

output\_mask.append(0)

input\_ids = torch.LongTensor(input\_ids).view(1, -1)

input\_mask = torch.BoolTensor(input\_mask).view(1, -1)

output\_mask = torch.BoolTensor(output\_mask).view(1, -1)

predict = model.infer(input\_ids, input\_mask, output\_mask)[0]

for i in range(len(test)):

print(test[i], end='', file=output)

print(id2tag[predict[i+1]], end=' ', file=label\_output)

if id2tag[predict[i+1]] in ['E', 'S']:

print(' ', end='', file=output)

print(file=output)

print(file=label\_output)

output.close()

label\_output.close()

## mergy.py

import sys

from tqdm import tqdm

with open('data/cws\_res\_bert.txt', 'r', encoding='utf8') as cws:

with open('data/ner\_res.txt', 'r', encoding='utf8') as ner:

with open('data/test\_data.txt', 'r', encoding='utf8') as test:

with open('cws\_result.txt', 'w', encoding='utf8') as out:

for cws\_label, ner\_label, line in tqdm(zip(cws, ner, test)):

cws\_label.strip()

ner\_label.strip()

line.strip()

print(line)

print(cws\_label)

print(ner\_label)

cws\_label = cws\_label.split(" ")

ner\_label = ner\_label.split(" ")

length = len(line)

# print(cws\_label,'\n',ner\_label)

print(len(cws\_label), len(ner\_label))

assert len(cws\_label) == len(ner\_label)

assert len(cws\_label) == length

if len(sys.argv) > 2 and sys.argv[1] == 'on':

i = 0

while i < length:

if line[i] == '《':

print('title')

j = i + 1

i = i + 1

while line[j] != '》':

j = j + 1

j = j - 1

if j - i == 0:

cws\_label[i] = 'S'

break

cws\_label[i] = 'B'

while i < j:

i = i + 1

cws\_label[i] = 'M'

cws\_label[j] = 'E'

if ner\_label[i][0] == 'B':

# print('encounter entity')

j = i

needReplace = False

count = 0

# print(line)

while ner\_label[j][0] != 'E':

if cws\_label[j] == 'E':

count = count + 1

if count >= 2:

needReplace = True

break

j = j + 1

if cws\_label[i] == 'E' and count == 1:

needReplace = True

if needReplace:

print('replace')

if cws\_label[i] == 'M' or cws\_label[i] == 'E':

if cws\_label[i - 1] == 'B':

cws\_label[i - 1] = 'S'

elif cws\_label[i - 1] == 'M':

cws\_label[i - 1] = 'E'

cws\_label[i] == 'B'

elif cws\_label[i] == 'S':

cws\_label[i] = 'B'

i = i + 1

while ner\_label[i][0] != 'E':

cws\_label[i] = 'M'

i = i + 1

if cws\_label[i] == 'B' or cws\_label[i] == 'M':

if cws\_label[i + 1] == 'M':

cws\_label[i + 1] = 'B'

elif cws\_label[i + 1] == 'E':

cws\_label[i + 1] = 'S'

cws\_label[i] = 'E'

elif cws\_label[i] == 'S':

cws\_label[i] == 'E'

else:

i = j

i = i + 1

for i in range(len(line)):

print(line[i], end='', file=out)

if cws\_label[i] in ['E', 'S']:

print(' ', end='', file=out)

# print(file=out)

## Model.py

from math import e

import torch

import torch.nn as nn

from torchcrf import CRF

from torch.nn.utils.rnn import pack\_padded\_sequence, pad\_packed\_sequence

from transformers import BertModel, BertTokenizer

class CWS(nn.Module):

def \_\_init\_\_(self, vocab\_size, tag2id, embedding\_dim, hidden\_dim):

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

self.embedding\_dim = embedding\_dim

self.hidden\_dim = hidden\_dim

self.vocab\_size = vocab\_size

self.tag2id = tag2id

self.tagset\_size = len(tag2id)

self.tokenizer = BertTokenizer.from\_pretrained("hfl/chinese-roberta-wwm-ext")

self.word\_embeds = BertModel.from\_pretrained("hfl/chinese-roberta-wwm-ext")

self.lstm = nn.LSTM(embedding\_dim, hidden\_dim // 2, num\_layers=1,

bidirectional=True, batch\_first=True)

self.hidden2tag = nn.Linear(hidden\_dim, self.tagset\_size)

self.crf = CRF(4, batch\_first=True)

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

return (torch.randn(2, batch\_size, self.hidden\_dim // 2, device=device),

torch.randn(2, batch\_size, self.hidden\_dim // 2, device=device))

def \_get\_lstm\_features(self, input\_ids, input\_mask):

batch\_size = input\_ids.size(0)

embeds = self.word\_embeds(input\_ids=input\_ids, attention\_mask=input\_mask, output\_hidden\_states=True, output\_attentions=True)

# idx->embedding

all\_hidden\_states, all\_attention = embeds[-2:]

embeds = all\_hidden\_states[-2]

# LSTM forward

self.hidden = self.init\_hidden(batch\_size, input\_ids.device)

lstm\_out, self.hidden = self.lstm(embeds, self.hidden)

lstm\_feats = self.hidden2tag(lstm\_out)

return lstm\_feats

def forward(self, input\_ids, label, input\_mask, output\_mask):

# print(input\_ids)

# print(label)

# print(input\_mask)

# print(output\_mask)

emissions = self.\_get\_lstm\_features(input\_ids, input\_mask)

loss = -self.crf(emissions, label, output\_mask, reduction='mean')

return loss

def infer(self, input\_ids, input\_mask, output\_mask):

emissions = self.\_get\_lstm\_features(input\_ids, input\_mask)

return self.crf.decode(emissions, output\_mask)

## run.py

import pickle

import logging

import argparse

import os

from re import L

import numpy as np

import torch

from torch.utils.data import DataLoader, TensorDataset

from torch.optim import Adam

from model import CWS

from tqdm import tqdm

os.environ["CUDA\_VISIBLE\_DEVICES"] = "0"

def get\_param():

parser = argparse.ArgumentParser()

parser.add\_argument('--embedding\_dim', type=int, default=768)

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

parser.add\_argument('--max\_epoch', type=int, default=10)

parser.add\_argument('--batch\_size', type=int, default=8)

parser.add\_argument('--hidden\_dim', type=int, default=256)

parser.add\_argument('--cuda', action='store\_true', default=True)

parser.add\_argument('--model', type=str, default='hfl/chinese-roberta-wwm-ext')

return parser.parse\_args()

def set\_logger():

log\_file = os.path.join('save', 'log.txt')

logging.basicConfig(

format='%(asctime)s %(levelname)-8s %(message)s',

level=logging.DEBUG,

datefmt='%Y-%m%d %H:%M:%S',

filename=log\_file,

filemode='w',

)

console = logging.StreamHandler()

console.setLevel(logging.DEBUG)

formatter = logging.Formatter('%(asctime)s %(levelname)-8s %(message)s')

console.setFormatter(formatter)

logging.getLogger('').addHandler(console)

def entity\_split(x, y, id2tag, entities, cur):

start, end = -1, -1

for j in range(len(x)):

if id2tag[y[j]] == 'B':

start = cur + j

elif id2tag[y[j]] == 'M' and start != -1:

continue

elif id2tag[y[j]] == 'E' and start != -1:

end = cur + j

entities.add((start, end))

start, end = -1, -1

elif id2tag[y[j]] == 'S':

entities.add((cur + j, cur + j))

start, end = -1, -1

else:

start, end = -1, -1

def main(args):

use\_cuda = args.cuda and torch.cuda.is\_available()

with open('data/datasave.pkl', 'rb') as inp:

tag2id = pickle.load(inp)

id2tag = pickle.load(inp)

input\_ids\_l = pickle.load(inp)

label\_l = pickle.load(inp)

input\_mask\_l = pickle.load(inp)

output\_mask\_l = pickle.load(inp)

model = CWS(0, tag2id, args.embedding\_dim, args.hidden\_dim)

if use\_cuda:

model = model.cuda()

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

logging.debug('%s: %s, require\_grad=%s' % (name, str(param.shape), str(param.requires\_grad)))

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

random\_order = list(range(len(input\_ids\_l)))

np.random.shuffle(random\_order)

input\_ids\_l = [input\_ids\_l[i] for i in random\_order]

input\_mask\_l = [input\_mask\_l[i] for i in random\_order]

label\_l = [label\_l[i] for i in random\_order]

output\_mask\_l = [output\_mask\_l[i] for i in random\_order]

data\_size = len(input\_ids\_l)

test\_size = data\_size // 10

test\_input\_ids = torch.LongTensor(input\_ids\_l[:test\_size])

train\_input\_ids = torch.LongTensor(input\_ids\_l[test\_size:])

test\_input\_mask = torch.BoolTensor(input\_mask\_l[:test\_size])

train\_input\_mask = torch.BoolTensor(input\_mask\_l[test\_size:])

test\_label = torch.LongTensor(label\_l[:test\_size])

train\_label = torch.LongTensor(label\_l[test\_size:])

test\_output\_mask = torch.BoolTensor(output\_mask\_l[:test\_size])

train\_output\_mask = torch.BoolTensor(output\_mask\_l[test\_size:])

train\_dataset = TensorDataset(train\_input\_ids, train\_input\_mask, train\_label, train\_output\_mask)

test\_dataset = TensorDataset(test\_input\_ids, test\_input\_mask, test\_label, test\_output\_mask)

train\_data = DataLoader(

dataset=train\_dataset,

shuffle=True,

batch\_size=args.batch\_size,

drop\_last=False,

num\_workers=6

)

test\_data = DataLoader(

dataset=test\_dataset,

shuffle=False,

batch\_size=args.batch\_size,

drop\_last=False,

num\_workers=6

)

for epoch in tqdm(range(args.max\_epoch)):

step = 0

log = []

for input\_ids, input\_mask, label, output\_mask in train\_data:

if use\_cuda:

input\_ids = input\_ids.cuda()

label = label.cuda()

input\_mask = input\_mask.cuda()

output\_mask = output\_mask.cuda()

# forward

loss = model(input\_ids, label, input\_mask, output\_mask)

log.append(loss.item())

# backward

optimizer.zero\_grad()

loss.backward()

optimizer.step()

step += 1

if step % 100 == 0:

logging.debug('epoch %d-step %d loss: %f' % (epoch, step, sum(log)/len(log)))

log = []

# test

entity\_predict = set()

entity\_label = set()

with torch.no\_grad():

model.eval()

cur = 0

for input\_ids, input\_mask, label, output\_mask in test\_data:

if use\_cuda:

input\_ids = input\_ids.cuda()

label = label.cuda()

input\_mask = input\_mask.cuda()

output\_mask = output\_mask.cuda()

predict = model.infer(input\_ids, input\_mask, output\_mask)

for i in range(len(predict)):

length = len([input\_ids[i][j] for j in input\_mask[i] if j == 1])

entity\_split(input\_ids[i, :length],

predict[i],

id2tag,

entity\_predict,

cur)

entity\_split(input\_ids[i, :length],

label[i, :length],

id2tag,

entity\_label,

cur)

cur += length

right\_predict = [i for i in entity\_predict if i in entity\_label]

if len(right\_predict) != 0:

precision = float(len(right\_predict)) / len(entity\_predict)

recall = float(len(right\_predict)) / len(entity\_label)

logging.info("precision: %f" % precision)

logging.info("recall: %f" % recall)

logging.info("f1score: %f" % ((2 \* precision \* recall) / (precision + recall)))

else:

logging.info("precision: 0")

logging.info("recall: 0")

logging.info("f1score: 0")

model.train()

path\_name = "./save/model\_epoch" + str(epoch) + ".pkl"

torch.save(model, path\_name)

logging.info("model has been saved in %s" % path\_name)

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

set\_logger()

main(get\_param())