華中科技大学

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 202001

学　　号：U202015533

姓　　名：徐瑞达

指导教师：魏巍

报告日期：2023 年 6 月 5 日

计算机科学与技术学院

目 录

1 中文分词实现

1.1 问题描述

本实验需要实现中文分词任务，可以考虑基于统计、基于词典的分词方法等，也可以考虑使用 Bi-LSTM+CRF 的深度学习模型进行分词，通过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。

1.1.1 基于词典匹配的分词算法

基于词典匹配的分词算法依赖人工建立的词库（词典）进行，包括正向最大匹配法、逆向最大匹配法以及双向最大匹配法。

1.1.2 基于统计学习的分词算法

将中文分词视作序列标注任务，给句子中的每个词打上合适的标签即可完成分词任务，而本实验使用 BIES 标注：

B：代表该字是一个词的开头

I：代表该字在一个词的内部

E：代表该字是一个词的结尾

S：代表该字独立成词

使用上述标注方式对句子进行标注，例如：

B E / S / B E / B I E / B E

小明 / 在 / 中国 / 科学院 / 工作

序列标注方法可以使用 HMM 等传统统计方法或者 Bi-LSTM 等深度学习方法，本实验使用基于双向长短神经网络+条件随机场（Bi-LSTM+CRF）的中文分词模型。

1.2 基础模块

本实验采用 Bert+Bi-LSTM+CRF 模型实现，主要由数据处理、数据加载、构建模型、训练及验证、预测等模块组成。(如图 1-1 所示)

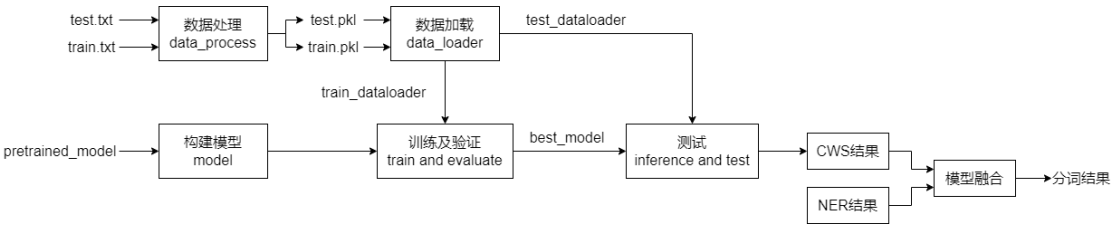

图 1-1 模块流程图

1.3 系统实现

1.3.1 总体实现

该系统可划分为数据处理、模型训练、模型预测三部分。本系统在 Bi-LSTM+CRF 的模型基础上，将数据编码部分替换为预训练模型 Bert，利用 Transfomer 的自注意力机制得到包含位置信息、语义信息的编码，从而获得更好的分词效果。在模型预测部分，根据 CWS 输出的标签序列和 NER 输出的标签序列进行模型融合，以减少 CWS 任务中对命名实体的错误划分。

run.py 文件为系统入口，首先加载预处理后的数据，并将其划分为训练集和验证集（此处划分比例为 9:1），然后从预训练模型中构建得到模型，设置优化率和学习率衰减，进行训练。每轮训练完成后，将得到的最优模型结果保存，当若干轮次的验证集损失不再有明显降低后即停止训练。根据最优模型结果，进行预测得到 CWS 任务的分词结果，然后使用同样的模型结构进行 NER 任务，得到命名实体标注结果，进行二者之间的模型融合得到最终分词结果。

1.3.2 数据处理

在 data_process.py 中，预处理训练集。在处理时，根据空格位置将每个字符转换为 BIES 标注。每行的标注结果存储在一个列表中，而由于 Bert 中输入序列的最长长度为 512 字符，因此当某行的长度大于 max_len 时，需要进行切分，切分后的多个子句需要在句尾添加分隔符以表示其属于同一个输入序列。

将预处理后的将预处理后的结果输出到二进制文件 training.pkl 中。

1.3.3 数据加载

在 data_loader.py 中，加载预处理后的数据到 Dataset 对象中。

首先从预训练模型 tacl-bert-base-chinese 中加载得到 BertTokenizer，将预处

理得到的 words_list 的每个字符转换为 token，并在每一行的开头加上[CLS]表示句子开头，每一句的结尾加上[SEP]表示句子分隔，经过以上处理后得到 dataset。

其次在 collate_fn 函数中获取 batch 数据时，还需要进行数据对齐。首先将每行数据填充至统一长度，然后将 token 与 label 对齐，最后将数据转换为张量。

1.3.4 构建模型

在 model.py 中，构建模型结构。模型依次分为 BertModel、Bi-LSTM、CRF 三层，首先根据输入的 token 与 label 获取 bert 模型的输出，然后去除[CLS]等标签，依次输入到 Bi-LSTM、CRF 获取损失值和输出。

BertModel 使用了 HuggingFace[2]提供的 transfomers 库中的 bert 模型，使用到的预训练模型有 bert-base-chinese、chinese-macbert-base、tacl-bert-base-chinese、RoBERTa-wwm-ext-large-chinese、LTP-tiny、LTP-small、LTP-base 等模型，最终结合模型性能和训练算力限制使用了 tacl-bert-base-chinese 模型。

1.3.5 训练及验证

在 train.py 中，进行模型的训练和验证。在每一轮训练过程中，依次进行模型预测、计算损失、反向传播、梯度裁剪、学习率衰减等过程，每轮训练完成后，即再验证集上评估当前模型性能，计算得到验证集损失和 F1-score。每得到更好的 F1-score 值，即保存模型参数，当 F1-score 变化不大时即停止训练。

1.3.6 模型预测与融合

在 test.py 中，进行 CWS 任务的模型预测。在 merge.py 中，进行 CWS 任务和 NER 任务结果的模型融合。进行模型融合时，先需要将 CWS 结果和 NER 结果输出为标注格式的文件；然后遍历两个文件，如果某个字符的 NER 标注为 I 则说明该字符属于某个命名实体，直接输出字符，否则根据 CWS 标注结果决定是否要在字符后输出空格。进行模型融合后，在线测试性能提升了 0.2 个百分点。

1.4 实验小结

1. 经过模型训练，最终选取的主要超参数如下：

lr = 0.005

epoch = 14

clip_grad = 5

batch_size = 32

hidden_dim = 200

embedding_dim = 100

weight_decay = 0.01

pretained_model = tacl-bert-base-chinese

最终得到的最高 F1 值为 0.912，相较基础模型提高了约 6 个百分点。

2. 训练过程中的遇到的问题如下：

由于微调预训练模型需要较大的算力，因此需要选择合适的预训练模型，对于 LTP-tiny、LTP-small 等模型，由于其参数规模较小，往往耗费算力训练十余个 epoch 之后仍较欠拟合，而对于参数规模较大的模型，其训练时间又会较长。经过综合考量，选取了 tacl-bert-base-chinese 模型，其使用了 TaCL (Token-aware Contrastive Learning)技术[3]，能够得到更具各向同性和判别性的语义信息。

3. 小结

在本实验中，通过预训练语言模型引入更好的语义编码，大大提升了中文分词的准确性。通过对预训练语言模型在训练集上进行微调，即可使其在下游任务上得到很好的表现。然而，表现较好的预训练语言模型参数规模也较大，微调时使用的算法也较大，因此，开发轻量级的预训练语言模型是 NLP 领域的重要研究方向。

通过本次实验，我了解并掌握了 NLP 任务的处理流程——数据处理、模型构建、训练优化、模型测试等步骤，也让我对 NLP 领域产生了更加浓厚的兴趣。

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

[2] Hugging Face.https://huggingface.co/

[3] Yixuan Su, Fangyu Liu, Zaiqiao Meng, Tian Lan, Lei Shu, Ehsan Shareghi, Nigel Collier. TaCL: Improving BERT Pre-training with Token-aware Contrastive Learning

附录 A 中文分词实现的源程序

data_process.py

```
import os
import pickle
import re
import config

class Processor:
    """
    数据处理器
    """

    def process(self):
        self.process_data("train")
        self.process_data("test")

    @staticmethod
    def cut(raw, max_len, sep):
        """
        将 raw 按照 max_len 切分，分隔符为 sep_word
        """
        list_groups = zip(*(iter(raw),) * max_len)
        end_list = [list(i) + list(sep) for i in list_groups]
        count = len(raw) % max_len
        if count != 0:
            end_list.append(raw[-count:])
        else:
            end_list[-1] = end_list[-1][:-1]
        return end_list

    @staticmethod
    def toTag(input):
        """
        将每个字转换为 BMES 标注
        """
        output_str = []
        if len(input) == 1:
            output_str.append('S')
        elif len(input) == 2:
            output_str = ['B', 'E']
        else:
```

```
            M_num = len(input) - 2
            M_list = ['M'] * M_num
            output_str.append('B')
            output_str.extend(M_list)
            output_str.append('E')
        return output_str

    def process_data(self, mode):
        # 判断训练集或测试集结果是否已存在
        if mode == "train":
            path = config.TRAINING_PATH
            origin_path = config.TRAINING_DATA
        else:
            path = config.TESTING_PATH
            origin_path = config.TEST_DATA
        if os.path.exists(path) is True:
            return
        with open(origin_path, 'r', encoding='utf-8') as f:
            words_list = []
            tags_list = []
            # 行数
            count = 0
            # 切分次数
            sep_count = 0
            for line in f:
                words = []
                tags = []
                line = line.strip()
                if not line:
                    continue
                # 读取每个字到 words
                for i in range(len(line)):
                    if line[i] == " ":
                        continue
                    words.append(line[i])
                # 将每个字转换为 BIES 标注,记录在 labels
                line = line.split(" ")
                for item in line:
                    if item == "":
                        continue
                    tags.extend(Processor.toTag(item))
                # 如果字数大于设定的最大长度，则进行切分
                if len(words) > config.max_len:
                    # 获取切分后的 words 列表
```

```
                    sub_words_list = Processor.cut(words, config.max_len - 5, config.sep_word)
                    sub_labels_list = Processor.cut(tags, config.max_len - 5, config.sep_label)
                    words_list.extend(sub_words_list)
                    tags_list.extend(sub_labels_list)
                    sep_count += 1
                else:
                    words_list.append(words)
                    tags_list.append(tags)
                count += 1
            # 将结果输出为二进制
            with open(path, 'wb') as outp:
                pickle.dump(words_list, outp)
                pickle.dump(tags_list, outp)


class Parser:
    """
    数据读取器
    """

    def analysis(self, mode='training'):
        len_list, word_list = self.read_file(mode)
        lens = {'<100': 0, '100-200': 0, '200-500': 0, '500-1000': 0, '>1000': 0}
        print(len(len_list), "sentences in the", mode, "file.")
        for i in len_list:
            if i <= 100:
                lens['<100'] += 1
            elif 100 < i <= 200:
                lens['100-200'] += 1
            elif 200 < i <= 500:
                lens['200-500'] += 1
            elif 500 < i <= 1000:
                lens['500-1000'] += 1
            elif i > 1000:
                lens['>1000'] += 1
        return lens

    def read_file(self, mode='training'):
        """
        读取文件
        """
        word_list = []
```

```
    len_list = []
    if mode == "train":
        origin_path = config.TRAINING_DATA
    else:
        origin_path = config.TEST_DATA
    with open(origin_path, 'r', encoding='utf-8') as f:
        for line in f:
            words = []
            line = line.strip()
            if not line:
                continue
            for i in range(len(line)):
                if line[i] == " ":
                    continue
                words.append(line[i])
            if len(words) > config.max_len:
                sub_word_list = self.get_sep_list(words, config.sep_word)
                for wl in sub_word_list:
                    if len(wl) > config.max_len or len(wl) == 0:
                        continue
                    word_list.append(wl)
                    len_list.append(len(wl))
            else:
                word_list.append(words)
                len_list.append(len(words))
    return len_list, word_list

def get_sep_list(self, init_list, sep_word):
    """
    按标点切分
    """
    w = "".join(init_list)
    s = re.split(r"(。)", w)
    s = self.add_sep_word(s, sep_word)
    s.append("")
    s = ["".join(i) for i in zip(s[0::2], s[1::2])]
    r = []
    for sub_list in s:
        r.append(list(sub_list))
    return r

def add_sep_word(self, s_, sep_word):
    """
    add sep word to string
```

```
        """
        new = []
        for i, item in enumerate(s_):
            if item == "，" or item == "。" or item == "；":
                if i == len(s_) - 2:
                    if s_[-1] == '':
                        new.append(item)
                        continue
                item += sep_word
            new.append(item)
        s_ = new
        return s_


def run():
    if os.path.exists(config.TRAINING_PATH):
        os.remove(config.TRAINING_PATH)
    if os.path.exists(config.TESTING_PATH):
        os.remove(config.TESTING_PATH)
    Processor().process()


def anlysis():
    print(Parser().analysis("train"))
    print(Parser().analysis("test"))


if __name__ == "__main__":
    run()
    anlysis()
```

data_loader.py

```
import numpy as np
import torch
import pickle
from torch.utils.data import Dataset, DataLoader
from transformers import BertTokenizer

import config


class Sentence(Dataset):
    """
```

```
    数据集对象
    """

    def __init__(self, words, tags):
        self.tokenizer = BertTokenizer.from_pretrained(config.BERT_MODEL_PATH,
do_lower_case=True)
        self.dataset = self.preprocess(words, tags)
        self.word_pad_idx = 0
        self.label_pad_idx = -1

    def preprocess(self, origin_words_list, origin_tags_list):
        data = []
        words_list = []
        tags_list = []
        for origin_words in origin_words_list[0:1]:
            words = []
            words_len = []
            for token in origin_words:
                words.append(self.tokenizer.tokenize(token))
                words_len.append(len(token))
            words = ['[CLS]'] + [item for token in words for item in token]
            token_start_idxs = 1 + np.cumsum([0] + words_len[:-1])
            words_list.append(((self.tokenizer.convert_tokens_to_ids(words), token_start_idxs),
origin_words))
        for origin_tags in origin_tags_list[0:1]:
            tags = [config.tag2id.get(t) for t in origin_tags]
            tags_list.append(tags)
        for words, tags in zip(words_list, tags_list):
            data.append((words, tags))
        return data

    def __len__(self):
        return len(self.dataset)

    def __getitem__(self, idx):
        word = self.dataset[idx][0]
        label = self.dataset[idx][1]
        return [word, label]

    def collate_fn(self, batch):
        """
            生成 batch
                process batch data, including:
                    1. padding: 将每个 batch 的 data padding 到同一长度（batch 中最长的
```

data 长度）

2. aligning: 找到每个 sentence sequence 里面有 label 项，文本与 label 对齐

3. tensor：转化为 tensor

```
        """
        # 处理后句子
        sentences = [x[0][0] for x in batch]
        # 原始句子
        ori_sentences = [x[0][1] for x in batch]
        # 标签
        labels = [x[1] for x in batch]
        # batch 大小
        batch_len = len(sentences)
        # 最长的句子长度
        max_len = max([len(s[0]) for s in sentences])
        max_label_len = 0

        # 初始化对齐后数据
        batch_data = self.word_pad_idx * np.ones((batch_len, max_len))
        batch_label_starts = []

        # 对齐数据
        for j in range(batch_len):
            cur_len = len(sentences[j][0])
            batch_data[j][:cur_len] = sentences[j][0]

            word_indexs = sentences[j][-1]
            label_starts = np.zeros(max_len)
            # 标记 label 开始位置
            label_starts[[idx for idx in word_indexs if idx < max_len]] = 1
            batch_label_starts.append(label_starts)
            max_label_len = max(int(sum(label_starts)), max_label_len)

        # 初始化对齐后标签
        batch_labels = self.label_pad_idx * np.ones((batch_len, max_label_len))

        # 对齐标签
        for j in range(batch_len):
            cur_tags_len = len(labels[j])
            batch_labels[j][:cur_tags_len] = labels[j]
        # 转化为 tensor
        batch_label_starts = np.array(batch_label_starts)
        batch_data = torch.LongTensor(batch_data)
        batch_label_starts = torch.LongTensor(batch_label_starts)
```

```
        batch_labels = torch.LongTensor(batch_labels)
        return [batch_data, batch_label_starts, batch_labels, ori_sentences]


if __name__ == '__main__':
    with open(config.TRAINING_PATH, 'rb') as inp:
        words = pickle.load(inp)
        labels = pickle.load(inp)
    dataset = Sentence(words, labels)
    train_dataloader = DataLoader(dataset, batch_size=config.batch_size, shuffle=True,
                                  collate_fn=dataset.collate_fn)
    for batch_data, batch_label_starts, batch_labels, ori_sentences in train_dataloader:
        print(batch_data, batch_label_starts, batch_labels, ori_sentences)
        break
```

model.py

```
from torch import nn
from transformers.models.bert import *
from torch.nn.utils.rnn import pad_sequence
from torchcrf import CRF


class BertSeg(BertPreTrainedModel):
    def __init__(self, config):
        super(BertSeg, self).__init__(config)
        self.num_labels = config.num_labels
        self.bert = BertModel(config)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.bilstm = nn.LSTM(
            input_size=config.lstm_embedding_size,    # 768
            hidden_size=config.hidden_size // 2,    # 1024 / 2
            batch_first=True,
            num_layers=2,
            dropout=config.lstm_dropout_prob,    # 0.5
            bidirectional=True
        )
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)
        self.crf = CRF(config.num_labels, batch_first=True)
        self.init_weights()

    def forward(self, input_data, token_type_ids=None, attention_mask=None, labels=None,
                position_ids=None, inputs_embeds=None, head_mask=None):
        # input_data: (input_ids, input_token_starts)
```

```
        input_ids, input_token_starts = input_data
        # 获取 bert 模型的输出
        outputs = self.bert(input_ids,
                            attention_mask=attention_mask,
                            token_type_ids=token_type_ids,
                            position_ids=position_ids,
                            head_mask=head_mask,
                            inputs_embeds=inputs_embeds)
        sequence_output = outputs[0]
        # 去除[CLS]标签等位置，获得与 label 对齐的 pre_label 表示
        origin_sequence_output = [layer[starts.nonzero().squeeze(1)]
                                  for layer, starts in zip(sequence_output,
input_token_starts)]
        # 将 sequence_output 的 pred_label 维度 padding 到最大长度
        padded_sequence_output = pad_sequence(origin_sequence_output, batch_first=True)
        # dropout pred_label 的一部分 feature
        padded_sequence_output = self.dropout(padded_sequence_output)
        # 将 padded_sequence_output 输入到 bilstm 中
        lstm_output, _ = self.bilstm(padded_sequence_output)
        # 得到判别值
        logits = self.classifier(lstm_output)

        outputs = (logits,)
        if labels is not None:
            loss_mask = labels.gt(-1)
            # 计算损失
            loss = self.crf(logits, labels, loss_mask) * (-1)
            outputs = (loss,) + outputs
        return outputs
```

config.py

```
import os

# 文件路径
BASE_DIR = os.getcwd() + "/"
DATA_DIR = BASE_DIR + "data/"
MODEL_DIR = BASE_DIR + "model/"
BEST_MODEL_DIR = BASE_DIR + "model/best/"
MODEL_PREFIX = MODEL_DIR + "model_epoch"
LOG_PATH = BASE_DIR + "log.txt"
TRAINING_DATA = DATA_DIR + "train_pku.txt"
TEST_DATA = DATA_DIR + "test_data.txt"
TRAINING_PATH = DATA_DIR + "training.pkl"
```

```
TESTING_PATH = DATA_DIR + "testing.pkl"
BERT_MODEL_PATH = BASE_DIR + "pretrained/chinese-macbert-base/"
MODEL_PATH=MODEL_DIR+"mode.pkl"
# 超参数
lr = 0.005
max_epoch = 10
min_epoch = 5
clip_grad = 5
batch_size = 1
hidden_dim = 200
embedding_dim = 100
weight_decay = 0.01
# worker 数量
num_workers = 2
# 是否使用 gpu
cuda = True
# 是否对 Bert 进行微调
full_fine_tuning = True
# 每行最大长度
max_len = 500
# 切分分隔符
sep_word = '@'
sep_label = 'S'
# 训练集和验证集划分
train_test_split_size = 0.1
random_state = 43
# 是否重新处理数据
reprocess_data = False
patience=5
# 标注
tag2id = {'B': 0, 'M': 1, 'E': 2, 'S': 3}
id2tag = {_id: _label for _label, _id in list(tag2id.items())}
```

run.py

```
import pickle
import logging
from train import train

from utils import set_logger
from transformers import get_cosine_schedule_with_warmup

import config
import torch
```

```
from torch.utils.data import DataLoader
from data_process import Processor
from data_loader import Sentence
from model import BertSeg
from torch.optim import AdamW
from sklearn.model_selection import train_test_split

def run():
    # 设置日志
    set_logger()
    # 是否开启 cuda
    enable_cuda = config.cuda and torch.cuda.is_available()
    # 处理数据
    if config.reprocess_data:
        logging.info("Process data...")
        Processor().process()
    # 加载数据
    logging.info("Load data...")
    with open(config.TRAINING_PATH, 'rb') as inp:
        words = pickle.load(inp)
        labels = pickle.load(inp)
    # 划分训练集和验证集
    word_train, word_test, label_train, label_test = train_test_split(words, labels,

test_size=config.train_test_split_size,

random_state=config.random_state)
    # 训练集
    train_dataset = Sentence(word_train, label_train)
    train_dataloader = DataLoader(
        dataset=train_dataset,
        shuffle=True,
        batch_size=config.batch_size,
        collate_fn=train_dataset.collate_fn,
        drop_last=False,
        num_workers=config.num_workers
    )
    # 验证集
    test_dataset = Sentence(word_test, label_test)
    test_dataloader = DataLoader(
        dataset=test_dataset,
        shuffle=False,
        batch_size=config.batch_size,
        collate_fn=test_dataset.collate_fn,
```

```
        drop_last=False,
        num_workers=config.num_workers
    )
    train_size = len(train_dataset)
    # 加载模型
    logging.info("Load pretrained model...")
    model = BertSeg.from_pretrained(config.BERT_MODEL_PATH,
num_labels=len(config.tag2id))
    # 是否使用 cuda
    if enable_cuda:
        model = model.cuda()
    if config.full_fine_tuning:
        # 全部参数参与训练
        bert_optimizer = list(model.bert.named_parameters())
        classifier_optimizer = list(model.classifier.named_parameters())
        no_decay = ['bias', 'LayerNorm.bias', 'LayerNorm.weight']
        optimizer_grouped_parameters = [
            {'params': [p for n, p in bert_optimizer if not any(nd in n for nd in no_decay)],
             'weight_decay': config.weight_decay},
            {'params': [p for n, p in bert_optimizer if any(nd in n for nd in no_decay)],
             'weight_decay': 0.0},
            {'params': [p for n, p in classifier_optimizer if not any(nd in n for nd in no_decay)],
             'lr': config.lr * 5, 'weight_decay': config.weight_decay},
            {'params': [p for n, p in classifier_optimizer if any(nd in n for nd in no_decay)],
             'lr': config.lr * 5, 'weight_decay': 0.0},
            {'params': model.crf.parameters(), 'lr': config.lr * 5}
        ]
    else:
        # 只有分类器参与训练
        classifier_optimizer = list(model.classifier.named_parameters())
        optimizer_grouped_parameters = [{'params': [p for n, p in classifier_optimizer]}]
    # 优化器
    optimizer = AdamW(optimizer_grouped_parameters, lr=config.lr)
    # 训练步数
    train_steps_per_epoch = train_size // config.batch_size
    # 学习率衰减
    scheduler = get_cosine_schedule_with_warmup(optimizer,
                                                num_warmup_steps=2 *
train_steps_per_epoch,

num_training_steps=config.max_epoch * train_steps_per_epoch)
    # 训练
    train(train_dataloader,test_dataloader, model, optimizer, scheduler, enable_cuda)
```

```
if __name__ == '__main__':
    run()
```

test.py

```
import torch
import pickle
import config
from data_loader import Sentence
from torch.utils.data import DataLoader

from model import BertSeg

if __name__ == '__main__':
    # 加载模型
    model = BertSeg.from_pretrained(
        config.BERT_MODEL_PATH, num_labels=len(config.tag2id))
    model.eval()
    # 是否使用 cuda
    enable_cuda = config.cuda and torch.cuda.is_available()
    # 输出文件
    output = open('cws_result.txt', 'w', encoding='utf-8')
    # 加载测试集
    with open(config.TESTING_PATH, 'rb') as inp:
        word_test = pickle.load(inp)
        label_test = pickle.load(inp)
    # 测试集
    test_dataset = Sentence(word_test, label_test)
    test_loader = DataLoader(
        dataset=test_dataset,
        shuffle=False,
        batch_size=config.batch_size,
        collate_fn=test_dataset.collate_fn,
        drop_last=False,
        num_workers=config.num_workers
    )

    id2tag = config.id2tag
    pred_labels = []

    print("Start predict...")

    with torch.no_grad():
```

```
        for idx, batch_samples in enumerate(test_loader):
            batch_data, batch_token_starts, batch_labels, ori_data = batch_samples
            if enable_cuda:
                batch_data = batch_data.cuda()
                batch_token_starts = batch_token_starts.cuda()
                batch_labels = batch_labels.cuda()
            # 获取 mask
            batch_masks = batch_data.gt(0)
            label_masks = batch_labels.gt(-1)
            # 计算模型预测值
            batch_output = model((batch_data, batch_token_starts),
                                 token_type_ids=None, attention_mask=batch_masks)[0]
            batch_output = model.crf.decode(batch_output, mask=label_masks)
            pred_labels.extend([[id2tag.get(idx) for idx in indices]
                                for indices in batch_output])

    print("Start output...")
    # 输出结果
    with open(config.TEST_DATA, 'r', encoding='utf-8') as f:
        for idx, test in enumerate(f):
            if (idx < pred_labels.__len__()):
                for i in range(len(test)):
                    print(test[i], end='', file=output)
                    if (i < pred_labels[idx].__len__() and pred_labels[idx][i] in ['E', 'S']):
                        print(' ', end='', file=output)
                print(file=output)
    print("Done!")
```

train.py

```
import logging
import torch
from tqdm import tqdm
import config
from torch import nn

from utils import f1_score

def train(train_dataloader,test_dataloader, model, optimizer, scheduler,enable_cuda):
    best_f1=0
    patience=0
    # 开始训练
    for epoch in range(config.max_epoch):
        # 训练
```

```
        train_epoch(train_dataloader, model, optimizer, scheduler, epoch, enable_cuda)
        # 验证
        result = evaluate(test_dataloader, model, enable_cuda)
        logging.info(
            "Epoch: {}, test loss: {}, fscore: {}, percision: {},recall: {}".format(epoch, result['loss'],

result['f'],

result['p'], result['r']))
        f1 = result['f']
        improve = f1 - best_f1
        if improve > 1e-5:
            best_f1 = f1
            model.save_pretrained(config.BEST_MODEL_DIR)
            if improve < config.patience:
                patience += 1
            else:
                patience = 0
        else:
            patience += 1
        if (patience >= config.patience and epoch > config.min_epoch) or epoch == config.max_epoch:
            break
        # 保存模型
        # torch.save(model, config.MODEL_PREFIX + str(epoch) + ".pkl")

def train_epoch(train_dataloader, model, optimizer, scheduler, epoch, enable_cuda):
    """
    进行一轮训练
    :param train_dataloader: 数据
    :param model: 模型
    :param optimizer: 优化器
    :param scheduler: 学习率预热
    :param epoch: 轮数
    :param enable_cuda: 是否使用 cuda
    """
    torch.cuda.empty_cache()
    logging.info("Epoch: {} start...".format(epoch))
    model.train()
    train_losses = 0
    for idx, batch_samples in enumerate(train_dataloader):
        # 获取数据
        batch_data, batch_token_starts, batch_labels, _ = batch_samples
```

```
            # 是否使用 cuda
            if enable_cuda:
                batch_data = batch_data.cuda()
                batch_token_starts = batch_token_starts.cuda()
                batch_labels = batch_labels.cuda()
            # 获取 mask
            batch_masks = batch_data.gt(0)
            # 模型预测
            loss = model((batch_data, batch_token_starts),
                         token_type_ids=None, attention_mask=batch_masks,
labels=batch_labels)[0]
            # 计算损失
            train_losses += loss.item()
            # 梯度清零
            model.zero_grad()
            # 反向传播
            loss.backward()
            # 梯度裁剪
            nn.utils.clip_grad_norm_(parameters=model.parameters(), max_norm=config.clip_grad)
            # 更新参数
            optimizer.step()
            # 更新学习率
            scheduler.step()
            if idx%100==0:
                logging.info("Epoch: {}, step: {}".format(epoch, idx))
        # 计算平均损失
        train_loss = float(train_losses) / len(train_dataloader)
        logging.info("Epoch: {}, train loss: {}".format(epoch, train_loss))


def evaluate(test_dataloader, model, enable_cuda):
        """
        评估模型性能
        :param test_dataloader: 验证集数据
        :param model: 模型
        :param enable_cuda: 是否使用 cuda
        """
        # 模型评估
        model.eval()
        id2tag = config.id2tag
        true_labels = []
        pred_labels = []
        test_losses = 0
        # 不计算梯度
```

```
    with torch.no_grad():
        for batch_samples in test_dataloader:
            batch_data, batch_token_starts, batch_labels, ori_data = batch_samples
            # 是否使用 cuda
            if enable_cuda:
                batch_data = batch_data.cuda()
                batch_token_starts = batch_token_starts.cuda()
                batch_labels = batch_labels.cuda()
            # 获取 mask
            batch_masks = batch_data.gt(0)
            label_masks = batch_labels.gt(-1)
            # 模型预测
            loss = model((batch_data, batch_token_starts),
                         token_type_ids=None, attention_mask=batch_masks,
labels=batch_labels)[0]
            # 计算损失
            test_losses += loss.item()
            # 获取预测结果
            batch_output = model((batch_data, batch_token_starts),
                                 token_type_ids=None, attention_mask=batch_masks)[0]
            # 解码
            batch_output = model.crf.decode(batch_output, mask=label_masks)
            batch_labels = batch_labels.to('cpu').numpy()

            pred_labels.extend([[id2tag.get(idx) for idx in indices] for indices in batch_output])
            true_labels.extend([[id2tag.get(idx) for idx in indices if idx > -1] for indices in
batch_labels])

    # 计算 f1 值
    res = {}
    f, p, r = f1_score(true_labels,pred_labels)
    res['f'] = f
    res['p'] = p
    res['r'] = r
    res['loss'] = float(test_losses) / len(test_dataloader)
    return res
```

utils.py

```
import transformers
import config
import os
import logging
```

```
def set_logger():
    """
    设置日志
    """
    transformers.logging.set_verbosity_error()
    log_file = os.path.join(config.MODEL_DIR, config.LOG_PATH)
    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 get_entities(seq):
    """
    Gets entities from sequence.
    """
    # for nested list
    if any(isinstance(s, list) for s in seq):
        seq = [item for sublist in seq for item in sublist + ['O']]
    prev_tag = 'O'
    begin_offset = 0
    chunks = []
    for i, chunk in enumerate(seq + ['O']):
        tag = chunk[0]
        if end_of_chunk(prev_tag, tag):
            chunks.append((begin_offset, i - 1))
        if start_of_chunk(prev_tag, tag):
            begin_offset = i
        prev_tag = tag

    return chunks


def end_of_chunk(prev_tag, tag):
    """Checks if a chunk ended between the previous and current word.
```

```
    Args:
        prev_tag: previous chunk tag.
        tag: current chunk tag.
    Returns:
        chunk_end: boolean.
    """
    chunk_end = False

    if prev_tag == 'S':
        chunk_end = True
    if prev_tag == 'E':
        chunk_end = True
    # pred_label 中可能出现这种情形
    if prev_tag == 'B' and tag == 'B':
        chunk_end = True
    if prev_tag == 'B' and tag == 'S':
        chunk_end = True
    if prev_tag == 'B' and tag == 'O':
        chunk_end = True
    if prev_tag == 'M' and tag == 'B':
        chunk_end = True
    if prev_tag == 'M' and tag == 'S':
        chunk_end = True
    if prev_tag == 'M' and tag == 'O':
        chunk_end = True

    return chunk_end


def start_of_chunk(prev_tag, tag):
    """Checks if a chunk started between the previous and current word.
    Args:
        prev_tag: previous chunk tag.
        tag: current chunk tag.
    Returns:
        chunk_start: boolean.
    """
    chunk_start = False

    if tag == 'B':
        chunk_start = True
    if tag == 'S':
        chunk_start = True
```

```
    if prev_tag == 'O' and tag == 'M':
        chunk_start = True
    if prev_tag == 'O' and tag == 'E':
        chunk_start = True
    if prev_tag == 'S' and tag == 'M':
        chunk_start = True
    if prev_tag == 'S' and tag == 'E':
        chunk_start = True
    if prev_tag == 'E' and tag == 'M':
        chunk_start = True
    if prev_tag == 'E' and tag == 'E':
        chunk_start = True

    return chunk_start


def f1_score(y_true, y_pred):
    true_entities = set(get_entities(y_true))
    pred_entities = set(get_entities(y_pred))
    nb_correct = len(true_entities & pred_entities)
    nb_pred = len(pred_entities)
    nb_true = len(true_entities)

    p = nb_correct / nb_pred if nb_pred > 0 else 0
    r = nb_correct / nb_true if nb_true > 0 else 0
    score = 2 * p * r / (p + r) if p + r > 0 else 0
    return score, p, r
```

merge.py

```
output_cws = open('cws_result2.txt', 'w', encoding='utf-8')
cws_label = open('cws_label.txt', 'r', encoding='utf-8')
ner_label = open('ner_label.txt', 'r', encoding='utf-8')
cws_label_lines=cws_label.readlines()
ner_label_lines=ner_label.readlines()
for i in range(len(cws_label_lines)):
    cws_label_lines[i]=cws_label_lines[i].replace('\n','')
    ner_label_lines[i]=ner_label_lines[i].replace('\n','')
    cws=cws_label_lines[i].split('  ')
    ner=ner_label_lines[i].split('  ')
    if (i==len(cws_label_lines)-1):
        next_ner=['','']
    else:
        next_ner=ner_label_lines[i+1].replace('\n','').split('  ')
```

```
if len(next_ner)<2:
    next_ner=["","]
if(len(cws)<2):
    print(file=output_cws)
else:
    if ner[1]=='I' and next_ner[1]=='I':
        print(cws[0],end="",file=output_cws)
    else:
        if cws[1]=='E' or cws[1]=='S':
            print(cws[0],end=' ',file=output_cws)
        else:
            print(cws[0],end="",file=output_cws)
```