# BiLSTM实战1

## 课程介绍

本节以句子级情感分类任务为例介绍使用BiLSTM进行建模，自然语言处理中情感分类任务是对给定文本进行情感倾向分类的任务，粗略来看可以认为其是分类任务中的一类。对于情感分类任务，目前通常的做法是先对词或者短语进行表示，再通过某种组合方式把句子中词的表示组合成句子的表示。最后，利用句子的表示对句子进行情感分类。BiLSTM是Bi-directional Long Short-Term Memory的缩写，是由前向LSTM与后向LSTM组合而成。两者在自然语言处理任务中都常被用来建模上下文信息。利用传统的LSTM对句子进行建模存在一个问题，无法编码从后到前的信息。在更细粒度的分类时，如对于强程度的褒义、弱程度的褒义、中性、弱程度的贬义、强程度的贬义的五分类任务需要注意情感词、程度词、否定词之间的交互。举一个例子，“这个餐厅脏得不行，没有隔壁好”，这里的“不行”是对“脏”的程度的一种修饰，而通过BiLSTM可以更好的捕捉双向的语义依赖。

## 实验概述

## 实验步骤

### 步骤3：验证及测试函数

验证函数的类似与训练函数，其过程为针对每一批次的验证数据采用训练的模型参数对验证集计算损失与精度。

|  |
| --- |
| def evaluate(config, model, data\_iter, test=False):  model.eval()  loss\_total = 0  # 存储验证集所有batch的预测结果  predict\_all = np.array([], dtype=int)  # 存储验证集所有batch的真实标签  labels\_all = np.array([], dtype=int)  with torch.no\_grad():  for texts, labels in data\_iter:  outputs = model(texts)  loss = F.cross\_entropy(outputs, labels)  loss\_total += loss  labels = labels.data.cpu().numpy()  predic = torch.max(outputs.data, 1)[1].cpu().numpy()  labels\_all = np.append(labels\_all, labels)  predict\_all = np.append(predict\_all, predic)  # 计算验证集准确率  acc = metrics.accuracy\_score(labels\_all, predict\_all)  # 如果是测试集的话 计算一下分类报告和混淆矩阵  if test:  report = metrics.classification\_report(labels\_all, predict\_all, target\_names=config.class\_list, digits=4)  # 计算混淆矩阵  confusion = metrics.confusion\_matrix(labels\_all, predict\_all)  return acc, loss\_total / len(data\_iter), report, confusion  # 返回准确率和每个batch的平均损失  return acc, loss\_total / len(data\_iter) |

在测试函数test()中，首先加载当前在验证集测试中损失最小的参数，然后计算模型参数在测试集上的精度，每个batch的平均损失 分类报告、混淆矩阵。

|  |
| --- |
| def test(config, model, test\_iter):  # test  # 加载使验证集损失最小的参数  model.load\_state\_dict(torch.load(config.save\_path))  model.eval()  start\_time = time.time()  # 计算测试集准确率，每个batch的平均损失 分类报告、混淆矩阵  test\_acc, test\_loss, test\_report, test\_confusion = evaluate(config, model, test\_iter, test=True)  msg = 'Test Loss: {0:>5.2}, Test Acc: {1:>6.2%}'  print(msg.format(test\_loss, test\_acc))  print("Precision, Recall and F1-Score...")  print(test\_report)  print("Confusion Matrix...")  print(test\_confusion)  time\_dif = get\_time\_dif(start\_time)  print("Time usage:", time\_dif) |

### 步骤2：数据预处理模块

在utils.py中，主要定义词句的数据处理模块，主要包括build\_vocab，build\_dataset，DatasetIterater，对应实现功能为构建词典/字典，构建数据集和迭代器以及使用预训练词/字向量对词/字嵌入矩阵进行初始化。

首先导入依赖以及参数定义

|  |
| --- |
| # coding: UTF-8  import os  import torch  import numpy as np  import pickle as pkl  from tqdm import tqdm  import time  from datetime import timedelta  MAX\_VOCAB\_SIZE = 10000 # 词表长度限制  UNK, PAD = '<UNK>', '<PAD>' # 未知字，padding符号 |

文本处理的一个重要步骤是将单词映射为数字编号，进而将一个句子改写成一个整数数组。然后在其基础上进行embedding等处理，最后送入到模型中。构建词典，完成单词到数字编号的映射，词典不能太大，太大会导致embedding过程中产生大量的计算。同样，词典也不能太小，太小会导致过多的词失去独立的意义。因此，根据词频构建词典是一个比较优秀的选择，该方法使得词频高的单词有独立的数字编号，词频低的单词共用一个编号。

|  |
| --- |
| def build\_vocab(file\_path, tokenizer, max\_size, min\_freq):  vocab\_dic = {}  with open(file\_path, 'r', encoding='UTF-8') as f:  # 遍历每一行  for line in tqdm(f):  lin = line.strip()#去掉首尾空白符  if not lin:#遇到空行 跳过  continue  content = lin.split('\t')[0]#text label；每一行以\t为切分，拿到文本  for word in tokenizer(content):#分词 or 分字,构建词或字到频数的映射 统计词频/字频  vocab\_dic[word] = vocab\_dic.get(word, 0) + 1  # 根据 min\_freq过滤低频词，并按频数从大到小排序，然后取前max\_size个单词  vocab\_list = sorted([\_ for \_ in vocab\_dic.items() if \_[1] >= min\_freq], key=lambda x: x[1], reverse=True)[:max\_size]  # 构建词或字到索引的映射 从0开始  vocab\_dic = {word\_count[0]: idx for idx, word\_count in enumerate(vocab\_list)}  # 添加未知符和填充符的映射  vocab\_dic.update({UNK: len(vocab\_dic), PAD: len(vocab\_dic) + 1})  return vocab\_dic |

在build\_dataset函数中，分别对训练集、验证集、测试集进行处理，返回字/词典 预处理好的训练集、验证集、测试集。在load\_dataset函数中，我们定义以下流程，首先开始读入文本，然后对文本进行分词/字，然后对分词进行长截短填操，将词/字转换为索引，不在词/字典中的 用UNK对应的索引代替。

|  |
| --- |
| def build\_dataset(config, ues\_word):  # 定义分词/字函数tokenizer,分为（word-level/character-level）  if ues\_word: #基于词 提前用分词工具把文本分开 以空格为间隔  tokenizer = lambda x: x.split(' ') # 以空格隔开，word-level  else:#基于字符  tokenizer = lambda x: [y for y in x] # char-level  # 构建词/字典  if os.path.exists(config.vocab\_path): #如果存在构建好的词/字典 则加载  vocab = pkl.load(open(config.vocab\_path, 'rb'))  else:#构建词/字典（基于训练集）  vocab = build\_vocab(config.train\_path, tokenizer=tokenizer, max\_size=MAX\_VOCAB\_SIZE, min\_freq=1)  # 保存构建好的词/字典  pkl.dump(vocab, open(config.vocab\_path, 'wb'))  print("Vocab size: {len(vocab)}") #输出词/字典大小  def load\_dataset(path, pad\_size=32):  contents = []  with open(path, 'r', encoding='UTF-8') as f:  for line in tqdm(f):#遍历每一行  lin = line.strip()#去掉首尾空白符  if not lin:#遇到空行 跳过  continue  content, label = lin.split('\t')#每一行以\t为切分  words\_line = []  token = tokenizer(content)#对文本进行分词/分字  seq\_len = len(token) #序列/文本真实长度（填充或截断前）  if pad\_size:#对分词进行长截短填  if len(token) < pad\_size:#文本真实长度比填充长度短  token.extend([PAD] \* (pad\_size - len(token)))  else: #文本真实长度比填充长度长  token = token[:pad\_size]#截断  seq\_len = pad\_size#把文本真实长度设置为填充长度  # word to id  for word in token:#将词/字转换为索引，不在词/字典中的 用UNK对应的索引代替  words\_line.append(vocab.get(word, vocab.get(UNK)))  contents.append((words\_line, int(label), seq\_len))  return contents # [([...], 0), ([...], 1), ...]  #分别对训练集、验证集、测试集进行处理 把文本中的词或字转换为词/字典中的索引  train = load\_dataset(config.train\_path, config.pad\_size)  dev = load\_dataset(config.dev\_path, config.pad\_size)  test = load\_dataset(config.test\_path, config.pad\_size)  return vocab, train, dev, test#返回字/词典 预处理好的训练集、验证集、测试集 |

在本项目中，我们自定义一个迭代器，将数据分成多个批次的数据单元，然后推送到device中进行计算。

|  |
| --- |
| class DatasetIterater(object):#自定义数据集迭代器  def \_\_init\_\_(self, batches, batch\_size, device):  self.batch\_size = batch\_size  self.batches = batches#构建好的数据集  self.n\_batches = len(batches) // batch\_size#得到batch数量  self.residue = False # 记录batch数量是否为整数  if len(batches) % self.n\_batches != 0:  self.residue = True#True表示不能整除  self.index = 0  self.device = device  def \_to\_tensor(self, datas):  # 转换为tensor 并 to(device)  x = torch.LongTensor([\_[0] for \_ in datas]).to(self.device)  y = torch.LongTensor([\_[1] for \_ in datas]).to(self.device)  # seq\_len为文本的实际长度（不包含填充的长度） 转换为tensor 并 to(device)  seq\_len = torch.LongTensor([\_[2] for \_ in datas]).to(self.device)  return (x, seq\_len), y  def \_\_next\_\_(self):  if self.residue and self.index == self.n\_batches:#当数据集大小不整除 batch\_size时，构建最后一个batch  batches = self.batches[self.index \* self.batch\_size: len(self.batches)]  self.index += 1  batches = self.\_to\_tensor(batches)#把最后一个batch转换为tensor 并 to(device)  return batches  elif self.index >= self.n\_batches:  self.index = 0  raise StopIteration  else:  batches = self.batches[self.index \* self.batch\_size: (self.index + 1) \* self.batch\_size]#把当前batch转换为tensor 并 to(device)  self.index += 1  batches = self.\_to\_tensor(batches)  return batches  def \_\_iter\_\_(self):  return self  def \_\_len\_\_(self):  if self.residue:  return self.n\_batches + 1  else:  return self.n\_batches  def build\_iterator(dataset, config):#构建数据集迭代器  iter = DatasetIterater(dataset, config.batch\_size, config.device)  return iter |