自然语言处理

期中项目：IMDB 电影评论文本分类

中山大学计算机学院　计算机科学与技术

19335174　施天予

目录

一、概述

本次期中项目的内容是文本的二分类问题。需要先通过数据清洗和文本预处理，然后尝试三种文本表示方法：TF、TF-IDF、word2vec，最后通过一种分类方法完成二分类。我选择的分类模型是 RNN 中的 LSTM，并在此基础上也尝试加入 Attention 机制，用 Pytorch 框架实现。最后对实验结果和各种方法进行比较分析。

二、实验原理

1. TF

TF 是词频归一化后的概率表示，一个词语在某篇文档中出现的频率。一个词出现的越频繁，它的重要性就越高。公式如下, 单词 i 在文档 d 的 tf 值，表示为单词 i 在文档 d 中出现的次数除以文档 d 中单词的总数。

$$tf_{i,d} = \frac{n_{i,d}}{\sum_v n_{v,d}}$$

2. TF-IDF

仅仅用 TF 来表示一个词的重要性是不够的，某些词出现的频率很高，但是对于我们判断一个文档的类别没有什么用，因此我们需要把罕见词的权值提高，把常见词的权值降低，也就是逆文档频率 IDF。IDF 的计算公式如下，文档的总数 |D| 除以包含单词 i 的文档的数目。

$$idf_i = log\frac{|D|}{|\{j : t_i \in d_j\}|}$$

若一个单词 i 没有被任何文档包含，则会出现分母为 0 的情况，为了防止除以 0 现象的出现，我们将分母 +1，公式如下：

$$idf_i = log\frac{|D|}{|\{j : t_i \in d_j\}| + 1}$$

TF-IDF 综合了 TF 和 IDF，如果一个词在一篇文档中出现的频率高，并且在语料库中其他文档中出现的次数较少，则这个词对于该篇文档很重要，具有很好的类别区分能力。单词 i 的 TF-IDF 的值就是将 TF 和 IDF 进行相乘，某个词对于文章的重要性越高，它的 TF-IDF 值也越大，公式如下：

$$tfidf_{i,j} = tf_{i,j} \times idf_i$$

3. Word2Vec

Word2Vec，是一个做 Word Embedding 的模型，也就是说将单词从原始的数据空间映射到低维空间转为向量形式的模型。Word2Vec 是用一个一层的神经网络，把每个单词的稀疏词向量映射成为一个 n 维的稠密向量的过程，以达到文本降维的效果，可以用在许多文本分析任务的数据处理上。Word2Vec 里面有两个重要的模型，CBOW 模型 (Continuous Bag-of-Words

Model) 与 Skip-Gram 模型，CBOW 模型是给定上下文预测 input 单词，Skip-Gram 模型是给定 input 单词预测上下文。在 Word2Vec 中，还可以使用负采样 (Negative Sampling)，霍夫曼 (Hierarchical) 等方法完成加速。总而言之，对 Word2Vec 模型属于一个文档，它的输出是文档中每个单词的 n 维向量。

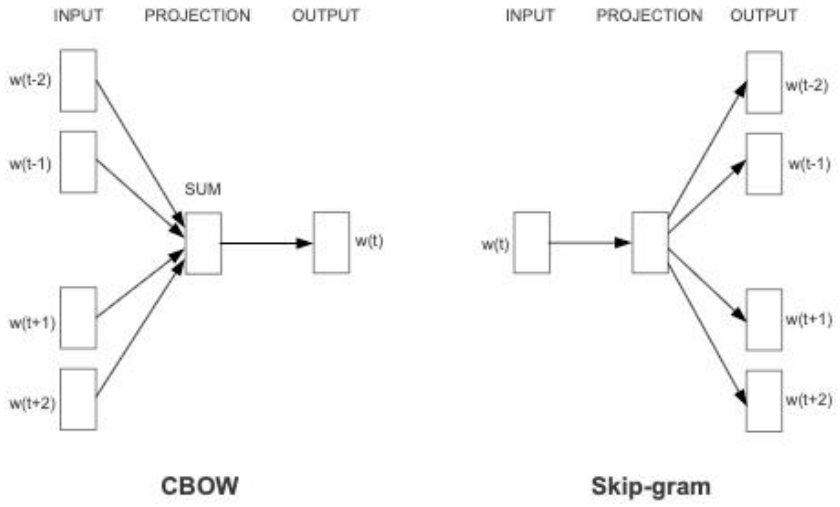

图 1: Word2Vec 模型

4. LSTM

对于一个情感二分类问题，我们使用的 RNN 循环神经网络是一种`Sequence-to-Vector`结构，即输入一个序列，输出一个单独的数据，通常在最后一个序列上进行输出变换。

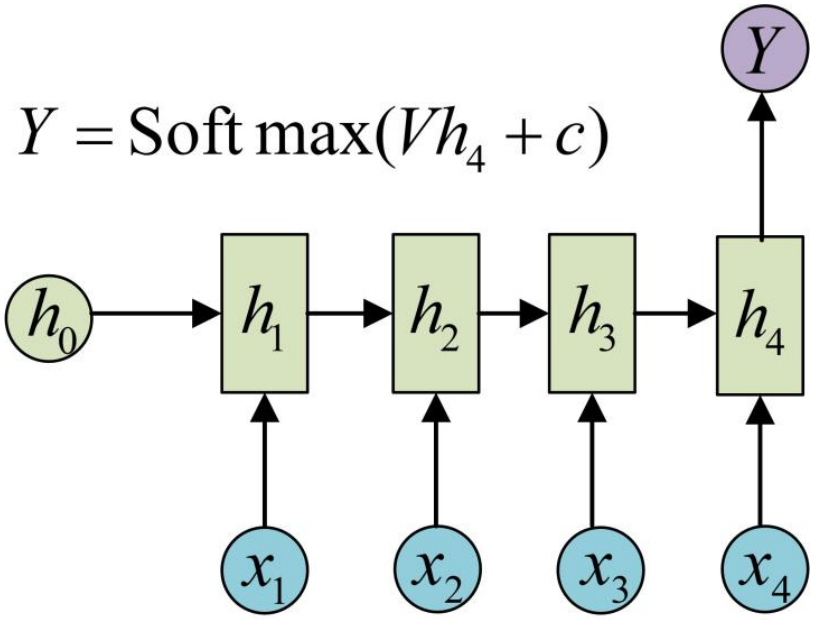

图 2: Sequence-to-Vector 结构

然而传统的 RNN 结构共享一组参数，梯度在反向传播过程中不断连乘，数值不是越来越大就是越来越小，就会出现**梯度爆炸**或是**梯度消失**的情况。LSTM 长短期记忆网络解决了这个问题，其神经元加入了输入门、遗忘门、输出门和内部记忆单元。模型结构如图3所示。

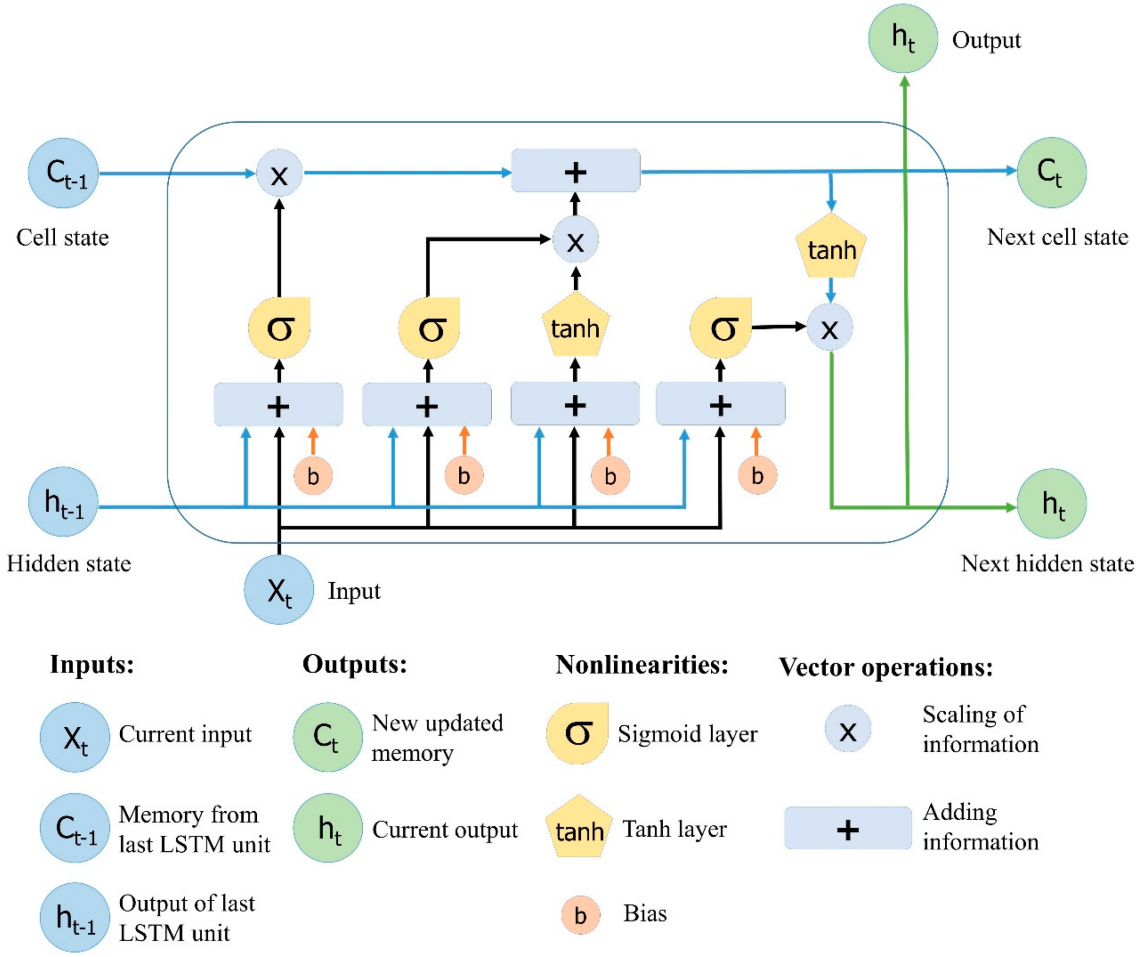

图 3: LSTM 单元

5. Attention 机制

在此基础上我也尝试了 LSTM+Attention 的模型结构。Attention 注意力机制，其实就是对之前 LSTM 的结果求出注意力概率分布，可以关注更重要的信息，使用 Attention 加权求和，生成最后结果，相比之前可以编码更长的序列信息。模型整体如图4所示：

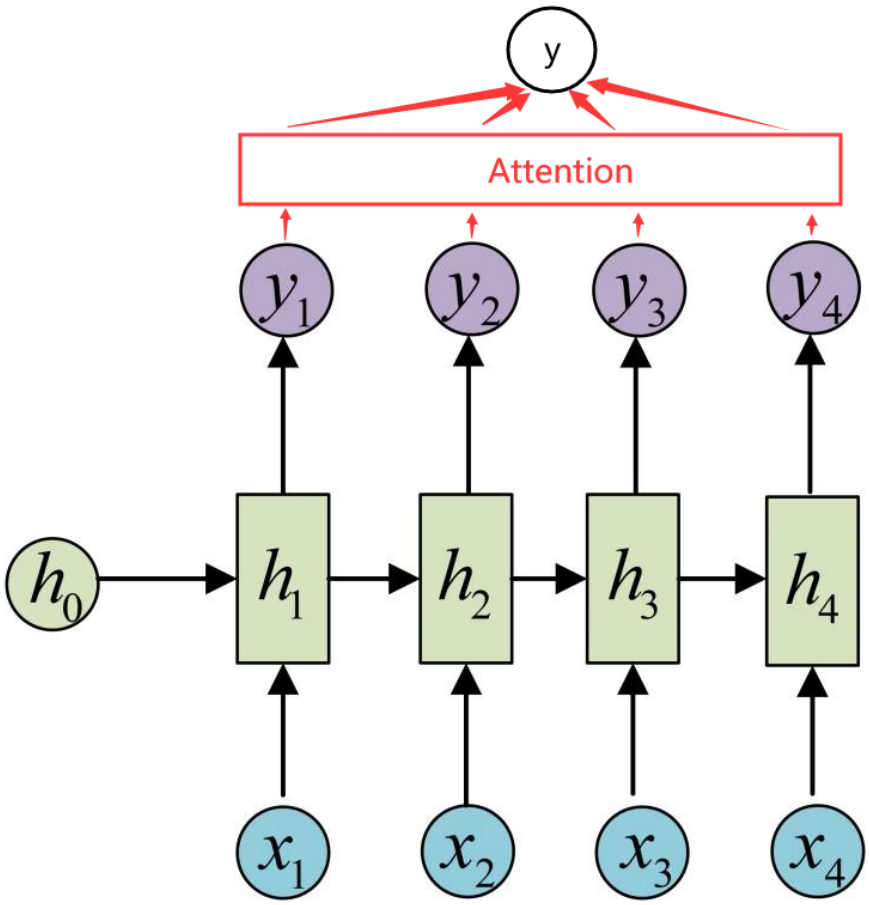

图 4: LSTM+Attention 模型

三、数据清洗和文本预处理

原始的文本数据是爬虫得到的初步结果，里面包含了 HTML 标签和 URL 等与文本无关的噪音，还有各种无用的符号。比如下面这个句子：

> Basically there's a family where a little boy (Jake) thinks there's a zombie in his closet & his parents are fighting all the time.

因此，需要进行数据清洗。可以去除 HTML 标记、消除无关字符、统一大小写等等。另外，对于一个文本分类问题，使用去除停用词、词形还原、词干提取等方法能更好地理解文本的语义。

1. 去除 HTML 标签

这里我使用了 BeautifulSoup 的 html.parser，帮助去除一系列 HTML 标签。

```
def strip_html(text):
    soup = BeautifulSoup(text, "html.parser")
    return soup.get_text()
data['review']=data['review'].apply(strip_html)
```

2. 消除无关字符及大小写统一

借助 ToktokTokenizer 分词工具，我们对句子中的每个词语用judge_pure_english函数消除无关字符，只保留英文字符，并且在消除无关字符的同时也用.lower() 的方法完成了英文字母大小写的统一。

```
tokenizer=ToktokTokenizer()
def judge_pure_english(word):
    return all(((ord(c)>64 and ord(c)<91) or (ord(c)>96 and ord(c)<128)) for c in word)
def remove_special_characters(text):
    tokens = tokenizer.tokenize(text)
    tokens = [token.strip() for token in tokens]
    filtered_tokens = [token.lower() for token in tokens if judge_pure_english(token)]
    filtered_text = ' '.join(filtered_tokens)
    return filtered_text
data['review']=data['review'].apply(remove_special_characters)
```

3. 去除停用词

停用词是一些对于文本语义分析没有用的单词，它们在每个句子中都会出现，所以需要去除来更好地理解文本。我使用了 nltk.corpus 中的“英文停用词”来进行去除，具体停用词如图5所示。

```
{'those', 'she', 'should', 'you', 'won', "hadn't", 'will', 'whom', 'yourself', 'y', 'below', 'down', "shan't", 'we', 'your', 'of
f', 'haven', 'were', 'only', "doesn't", 'no', 'from', "couldn't", 'of', "hasn't", 'very', "she's", 'hers', 'hadn', 'wasn', 'oursel
ves', "won't", 'what', 'are', 'a', 'its', 'o', 'been', 'once', 'mustn', 'after', 'not', "that'll", 'this', 'by', 'shan', "should
n't", 'themselves', 't', 'was', 'me', 'herself', 'ma', "mightn't", 'shouldn', 'if', 'and', 're', "needn't", "you'll", 'these', 'sa
me', 'as', 'with', 'mightn', 'don', 'hasn', 'their', 'so', 'being', 's', 'itself', 'theirs', 'own', 'did', 'do', 've', 'didn', 'th
ere', 'few', "isn't", 'each', 'because', 'the', "wouldn't", 'for', 'am', "you've", 'needn', "it's", 'through', 'can', "should've",
'does', 'just', 'other', "don't", 'over', 'has', 'above', 'any', 'd', 'm', 'couldn', 'out', 'is', 'll', 'nor', 'doing', 'yourselve
s', "you'd", 'an', "wasn't", "weren't", 'during', 'doesn', 'under', 'who', 'to', "didn't", 'be', 'up', 'in', 'here', 'some', 'whic
h', 'having', 'more', 'now', 'ain', 'while', 'that', 'on', 'then', 'her', 'how', 'such', 'when', 'all', 'too', 'before', 'have',
'my', 'than', 'i', 'into', 'yours', 'until', 'about', "you're", 'they', 'had', 'our', 'again', 'them', 'himself', 'ours', 'but',
"haven't", "aren't", 'against', 'wouldn', 'at', "mustn't", 'between', 'where', 'both', 'him', 'aren', 'he', 'weren', 'or', 'why',
'it', 'further', 'most', 'myself', 'isn', 'his'}
```

图 5: 停用词

因为这次的文本分类的主题是“电影”，“movie”和“film”在大部分文本都会出现，没有意义。所以我也把它们加入停用词一起去除。

```
stopwords = nltk.corpus.stopwords.words('english')
specific_sw = ['movie', 'film']
stop = set(stopwords+specific_sw)

def remove_stopwords(text):
    tokens = tokenizer.tokenize(text)
```

```
    tokens = [token.strip() for token in tokens]
    filtered_tokens = [token for token in tokens if token not in stop]
    filtered_text = ' '.join(filtered_tokens)
    return filtered_text
data['review']=data['review'].apply(remove_stopwords)
```

4. 词形还原

英文单词常常不是以原形出现的，如果能把单词的复数、过去式、过去分词等形式还原成原形，就能更好地理解单词的意思。对于词形还原，首先需要标注词语的类型，比如是形容词、动词、名词还是副词。然后根据词语本身和标注的标签，就能很好地完成词形还原了。

```
from nltk.corpus import wordnet
from nltk.stem import WordNetLemmatizer
# 获取单词的词性
def get_wordnet_pos(tag):
    if tag.startswith('J'):
        return wordnet.ADJ
    elif tag.startswith('V'):
        return wordnet.VERB
    elif tag.startswith('N'):
        return wordnet.NOUN
    elif tag.startswith('R'):
        return wordnet.ADV
    else:
        return None
# 词形还原
def lemmatization(text):
    wnl = WordNetLemmatizer()
    tokens = tokenizer.tokenize(text)
    tokens = [token.strip() for token in tokens]
    tagged_sent = nltk.pos_tag(tokens)
    lemmas_sent = []
    for tag in tagged_sent:
        wordnet_pos = get_wordnet_pos(tag[1]) or wordnet.NOUN
        lemmas_sent.append(wnl.lemmatize(tag[0], pos=wordnet_pos))
    sentence = ' '.join(lemmas_sent)
    return sentence
data['review']=data['review'].apply(lemmatization)
```

5. 词干提取

词形还原能把许多词语还原成原形，但我发现也有一定疏漏。比如“prepared”这个单词，如果在一个它是动词过去式的句子中把它辨别成形容词，那么就不会还原，因此我又使用了词干提取，对每个词语进一步化简。虽然词干提取后的词语并不一定是词语原形，甚至比原形更

短，不是我们常见的英文单词。但是对于计算机来说，词干提取能更好地理解语义。

```
from nltk.stem.porter import PorterStemmer
def stemming(text):
    ps = PorterStemmer()
    text = ' '.join([ps.stem(word) for word in text.split()])
    return text
data['review']=data['review'].apply(stemming)
```

四、文本表示方法

最开始我尝试自己实现了 TF 和 TF-IDF 的矩阵，但因为运行速度实在太慢，且不方便低频词的去除，所以后来我就调库实现了。下面是自己写的 TF 和 TF-IDF 代码：

```
# 构建词表
def count_words(train_data):
    word_list = []
    word_set = set()
    for line in train_data:
        words = line.split(' ')     # 将每一行用空格划分为多个单词
        for word in words:
            if word not in word_set: # 用集合加速查找
                word_list.append(word) # 如果单词不在词表中，则将单词加入词表
                word_set.add(word)
    return word_list
# 计算TF矩阵，求出TF矩阵的一行
def TF(words):
    tf = {}
    num = len(words) # 计算一行中的单词个数
    for word in words:
        tf[word] = (tf.get(word, 0)*num+1)/num # 按照公式计算TF
    return tf
# 计算IDF向量
def IDF(data, word_list):
    total = len(data)
    idf = []
    for word in word_list: # 遍历词表中的每个词
        count = 0
        for sentence in data:
            words = sentence.split(' ') # 将每一行用空格划分为多个单词
            if word in words:
                count += 1 # 如果该单词在这一行中，计数器count加上1
        idf.append(math.log(total/(count+1))) # 利用公式计算
    return idf
# 计算TF-IDF矩阵
def TFIDF(data, word_list):
```

```
    idf = IDF(data, word_list) # 计算idf的值
    tfidf = []
    for sentence in data:
        words = sentence.split(' ')
        tf = TF(words)          # 计算每一行的tf的值
        line = []
        index = 0
        for index, word in enumerate(word_list):
            x = tf.get(word, 0)*idf[index] # 利用公式计算tf-idf
            line.append(x)              # 在该行中添加一个值
        tfidf.append(line)      # tfidf矩阵添加一行
    tfidf = np.array(tfidf)
    return tfidf
```

后面的文本表示都通过调库实现。

1. TF

首先通过 CountVectorizer 工具将文本转为词语的计数矩阵，去除词频小于 5 的低频词。然后将每个句子每个词语的频数除以每个句子中的总单词个数，即可得到 TF 矩阵。

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer(min_df=5,dtype='float32')
# 先fit训练传入的文本数据，然后对文本数据进行标记并转换为稀疏计数矩阵
train_vectors = np.array(vectorizer.fit_transform(train_corpus).toarray())
valid_vectors = np.array(vectorizer.transform(valid_corpus).toarray())
test_vectors = np.array(vectorizer.transform(test_corpus).toarray())
# TF矩阵
for i in range(train_vectors.shape[0]):
    train_vectors[i] = train_vectors[i] / len(train_corpus[i].split())
for i in range(valid_vectors.shape[0]):
    valid_vectors[i] = valid_vectors[i] / len(valid_corpus[i].split())
for i in range(test_vectors.shape[0]):
    test_vectors[i] = test_vectors[i] / len(test_corpus[i].split())
```

2. TF-IDF

类似于 TF 矩阵，使用 CountVectorizer 工具将文本转为词语的计数矩阵后，再使用 TfidfTransformer 工具转为 TF-IDF 矩阵。同样去除词频小于 5 的低频词。

```
from sklearn.feature_extraction.text import TfidfTransformer
transform = TfidfTransformer()
train_vectors = np.array(transform.fit_transform(train_counts).toarray())
valid_vectors = np.array(transform.transform(valid_counts).toarray())
test_vectors = np.array(transform.transform(test_counts).toarray())
```

3. Word2Vec

Word2Vec 模型首先需要训练集的 30000 个样本作为语料库传入训练，设置词向量维度为 100，去除词频小于 5 的低频词，词语间窗口为 2。通常来说，对于上万样本甚至百万样本的训练集 epochs 设置为 10 到 20 比较合理，所以这里我也将 epochs 设为 20。

```
model = Word2Vec(sentences=train_corpus,vector_size=100, min_count=5, workers = 4,
    ↪ window = 2)
model.save('word2vec.model')
model.train(train_corpus, total_examples=model.corpus_count, epochs=20)
```

但是 Word2Vec 模型得到的是每个词语的词向量，那么如何表示一个句子呢？这里我尝试了两种方法。

(i) 预训练词向量取平均

首先我使用的是将一个句子中每个出现的单词的词向量加起来，再除以句子中词语的总数。这个方法虽然能表示一个句子的含义，但效果并不好，这在后面的部分中会有分析。

```
def make_sentence(corpus, model, num_features):
    vectors = []
    index2word_set = set(model.wv.index_to_key)
    for line in corpus:
        featureVec = np.zeros(num_features,dtype='float32')
        nwords = 0
        for word in line:
            if word in index2word_set:
                nwords = nwords + 1
                featureVec = np.add(featureVec,model.wv[word])
        featureVec = np.divide(featureVec, nwords)
        vectors.append(list(featureVec))
    return np.array(vectors)
train_vectors = make_sentence(train_corpus, model, model.vector_size)
valid_vectors = make_sentence(valid_corpus, model, model.vector_size)
test_vectors = make_sentence(test_corpus, model, model.vector_size)
```

(ii) 预训练词向量 Embedding

另外就是使用 nn.Embedding 将 word2vec 模型作为预训练词向量嵌入。这个方法需要给 word2vec 模型中的每个单词一个编号，然后形成一个 id 对应 vector 的 weight 矩阵，作为词嵌入矩阵。相应地，也要将数据集中的每个 word 转为对应的 id，验证集和测试集中出现训练集没有的单词就用 0 表示，这里 weight 矩阵的 0 行就是一个 100 维全 0 的向量。

```
w2v = Word2Vec.load('word2vec.model')
vocab = set(w2v.wv.index_to_key)
vocab_list = w2v.wv.index_to_key
```

```
vocab_list.insert(0,0)
vocab_size = len(vocab_list)
embed_size = 100
weight = np.zeros((vocab_size, embed_size),dtype='float32')
for i in range(1,vocab_size):
    weight[i,:] = w2v.wv[vocab_list[i]]

def w2i(corpus, vocab_list, vocab):
    vectors = []
    for line in corpus:
        vector = []
        for word in line:
            if word in vocab:
                vector.append(vocab_list.index(word, 0, len(vocab_list)))
        vectors.append(vector)
train_vectors = w2i(train_corpus, vocab_list, vocab)
valid_vectors = w2i(valid_corpus, vocab_list, vocab)
test_vectors = w2i(test_corpus, vocab_list, vocab)
```

需要注意的是，我们需要统一每一个句子的 word2id 矩阵的长度才能送到 LSTM 模型中完成训练。那么如何统一句子的长度呢？我先写了下面这段代码来统计训练集中语料库中每个句子的长度，结果如图6所示。

```
rev_len = [len(i) for i in train_vectors]
pd.Series(rev_len).hist()
plt.show()
pd.Series(rev_len).describe()
```

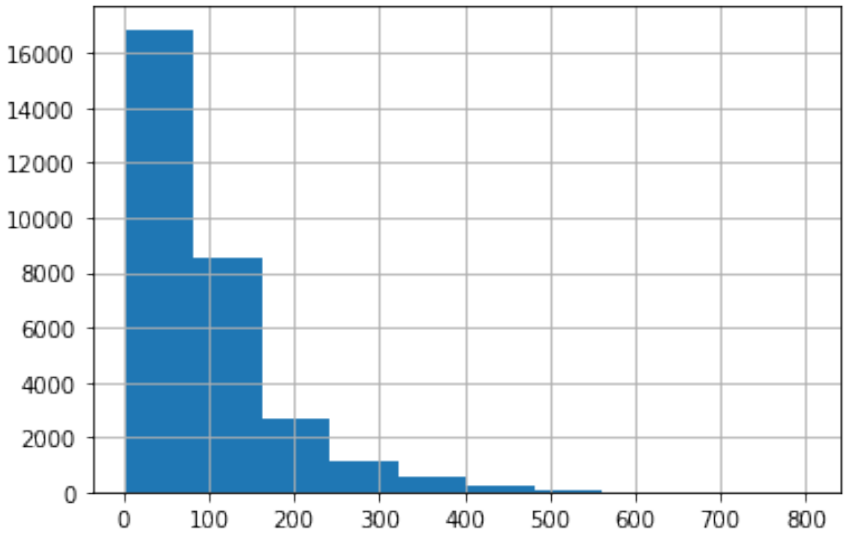

图 6: 训练集语料库中每个句子的长度

发现 95% 以上的句子长度都小于 400，因此选择 400 作为句子长度的统一值。为统一每个句子的长度，我们需要对短句子补 0 并且对长句子截断，这样就能得到一个表示数据集的 400 维矩阵。

```
def padding(sentences, seq_len):
    features = np.zeros((len(sentences), seq_len),dtype=int)
    for i, review in enumerate(sentences):
        if len(review) > seq_len:
            features[i, :] = np.array(review)[:seq_len]
        else:
            features[i, -len(review):] = np.array(review)[:seq_len]
    return features

train_vectors = padding(train_vectors, 400)
valid_vectors = padding(valid_vectors, 400)
test_vectors = padding(test_vectors, 400)
```

五、模型方法

1. 模型搭建

(i) 不用 nn.Embedding 的 LSTM 模型

对于 TF、TF-IDF、Word2Vec 词向量取平均的表示方法来说，其单个向量就已经表示了一个句子的语义，所以就不需要 nn.Embedding 再做词嵌入。然而 LSTM 输入的 Tensor 要求的形状是 (batch, time_step, input_size) 所以输入时需要将二维的 Tensor 先转为三维形式，只需要将第二维设为 1 即可。在 LSTM 层之后加入 dropout 防止过拟合，最后通过一个线性层输出结果。

```
class RNN(nn.Module):
    def __init__(self):
        super(RNN, self).__init__()
        self.rnn = nn.LSTM(
            input_size=EMBEDDING_DIM,
            hidden_size = HIDDEN_SIZE,
            num_layers = 1,
            batch_first = True,
        )
        self.dropout = nn.Dropout(0.5)
        self.out = nn.Linear(HIDDEN_SIZE, 2)
    def forward(self, x):
        x = x.view(len(x), 1, -1)
        r, _ = self.rnn(x)
        r = self.dropout(r)
        out = self.out(r[:, -1, :])
```

```
        return out
```

(ii) 使用 nn.Embedding 的 LSTM 模型

而对于使用 word2vec 模型做 nn.Embedding 词嵌入时，输入的二维 Tensor 第二维是句子的长度，在完成 nn.Embedding 层后就变成了一个三维 (batch, seq_len, input_size) 的 Tensor。需要注意将导入后的权重参数的requires_grad设置为 True，这样才能在之后的训练过程中迭代训练到这些参数，使得我们的 Embedding 层也可以根据数据分布进行微调。其他部分与之前相同。

```
class RNN(nn.Module):
    def __init__(self):
        super(RNN, self).__init__()
        self.word_embed = nn.Embedding.from_pretrained(weight)
        self.word_embed.weight.requires_grad = True
        self.rnn = nn.LSTM(
            input_size = EMBEDDING_DIM,
            hidden_size = HIDDEN_SIZE,
            num_layers = 1,
            batch_first = True,
        )
        self.dropout = nn.Dropout(0.5)
        self.out = nn.Linear(HIDDEN_SIZE, 2)

    def forward(self, x):
        r = self.word_embed(x)
        r, _ = self.rnn(r)
        r = self.dropout(r)
        out = self.out(r[:, -1, :])
        return out
```

(iii) LSTM+Attention 模型

在 LSTM 层之后，使用 Attention 机制先求出注意力概率分布，再对 LSTM 层的输出结果加权求和。理论上来说这样机器可以重点关注某些维度的特征，从而得到更好的结果。

```
class RNN(nn.Module):
    def __init__(self):
        super(RNN, self).__init__()
        self.word_embed = nn.Embedding.from_pretrained(weight)
        self.word_embed.weight.requires_grad = True
        self.rnn = nn.LSTM(
            input_size = EMBEDDING_DIM,
            hidden_size = HIDDEN_SIZE,
            num_layers = 1,
```

```
            batch_first = True,
        )
        self.w_omega = nn.Parameter(torch.Tensor(HIDDEN_SIZE, HIDDEN_SIZE))
        self.u_omega = nn.Parameter(torch.Tensor(HIDDEN_SIZE, 1))
        self.dropout = nn.Dropout(0.5)
        self.out = nn.Linear(HIDDEN_SIZE, 2)
        nn.init.uniform_(self.w_omega, -0.1, 0.1)
        nn.init.uniform_(self.u_omega, -0.1, 0.1)

    def att(self, x):
        u = torch.tanh(torch.matmul(x, self.w_omega))
        att = torch.matmul(u, self.u_omega)
        att_score = F.softmax(att, dim=1)
        scored_x = x * att_score
        feat = torch.sum(scored_x, dim=1) #加权求和
        return feat

    def forward(self, x):
        r = self.word_embed(x)
        r, _ = self.rnn(r)
        r = self.att(r)
        r = self.dropout(r)
        out = self.out(r)
        return out
```

2. 模型训练

接下来对模型进行训练。因为我已经使用训练集创建了 DataLoader，所以可以使用其来进行训练加载，训练过程定义如下：

```
for epoch in range(EPOCH):
    for step, (b_x, b_y) in enumerate(train_loader):
        rnn.train()
        b_x = b_x.to(device).long()
        b_y = b_y.to(device).long()
        output = rnn(b_x)

        loss = loss_func(output, b_y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

3. 模型评估

设置每个 epoch 计算一次模型在验证集的 loss 和准确率，如果当前验证集的准确率是最高的，那么就保存模型。使用`with torch.no_grad()`可以使模型在验证集测试时不计算梯度，从

而减少运行时间。

```
rnn.eval()
with torch.no_grad():
    valid_x = valid_x.to(device).long()
    valid_y = valid_y.to(device).long()
    valid_output = rnn(valid_x)
    valid_loss = loss_func(valid_output, valid_y)

    pred_train = output.argmax(-1)
    pred_valid = valid_output.argmax(-1)
    train_acc = (pred_train.cpu() == b_y.cpu()).float().mean().data.numpy()
    valid_acc = (pred_valid.cpu() == valid_y.cpu()).float().mean().data.numpy()

    if valid_acc > best_acc:
        best_acc = valid_acc
        torch.save(rnn.state_dict(), 'RNN2.ckpt')
    print('Epoch: ', epoch, '| train loss: %.4f' % loss.item(), '| train
        ↪ accuracy: %.4f' % train_acc, '| valid loss: %.4f' % valid_loss.item
        ↪ (), '| valid accuracy: %.4f' % valid_acc)
```

4. 参数调整

batch size

设置 batch size = **256**。在合理范围内增加 batch size，可以提高内存利用率，对于相同数据量的处理速度加快，并且一般来说 batch size 越大, 其确定的下降方向越准，引起的训练震荡越小。然而盲目增大 batch size 内存可能会撑不住，且想达到相同的精度需要更多时间，对参数的修正也更加缓慢。

optimizer

optimizer 优化器的选择上，使用 SGD 随机梯度下降需要手动指定学习率大小，如果太小会导致前期收敛速度很慢，而使用很大的学习率又会导致后期无法很好地收敛，所以我选择使用自适应学习率的优化器 **Adam**，可以快速收敛，其应用也非常广泛。

dropout

一般来说，防止过拟合的方法有 dropout、layer nomolization、batch nomolization 等。因为 batch nomolization 不适合于 RNN 这样的动态神经网络，而 layer nomolization 经检验发现会使模型效果更差，所以我使用了 dropout。dropout 可以在前向传播的时候，让某个神经元的激活值以一定的概率停止工作，使模型泛化性更强。这里我设置了 dropout = **0.5**

epochs

训练轮数的选择上，我发现在使用 Adam 优化器时，不需要太多的轮数就能基本收敛，为了避免过多的训练带来的过拟合，我设置 epochs = **10**。

batch size	256
embedding dim	100
hidden size	64
learning rate	0.001
epochs	10

表 1: 超参数设置

六、实验结果与分析

经过以上研究分析、代码编写、参数设置，使用 GPU 运行训练，记录每次迭代的 loss 和每次 epoch 训练集和验证集的准确率，结果如下：

1. TF

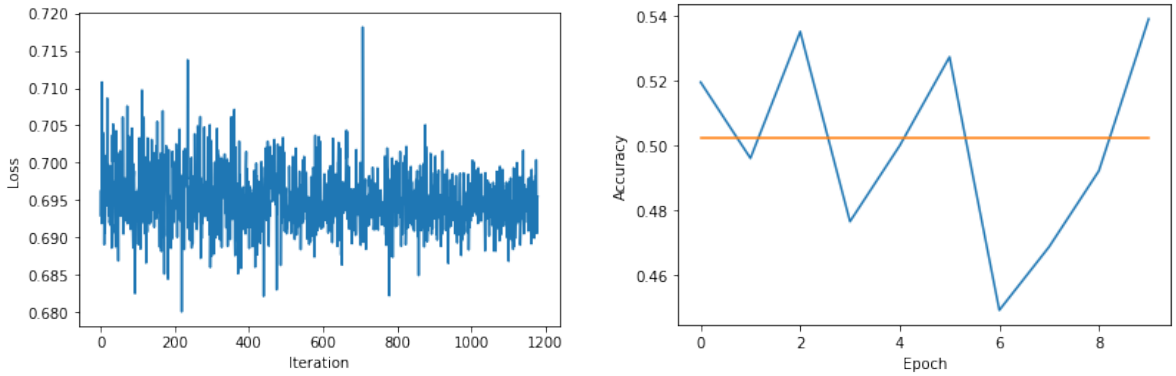

图 7: TF 在 SGD 下的 loss 和准确率

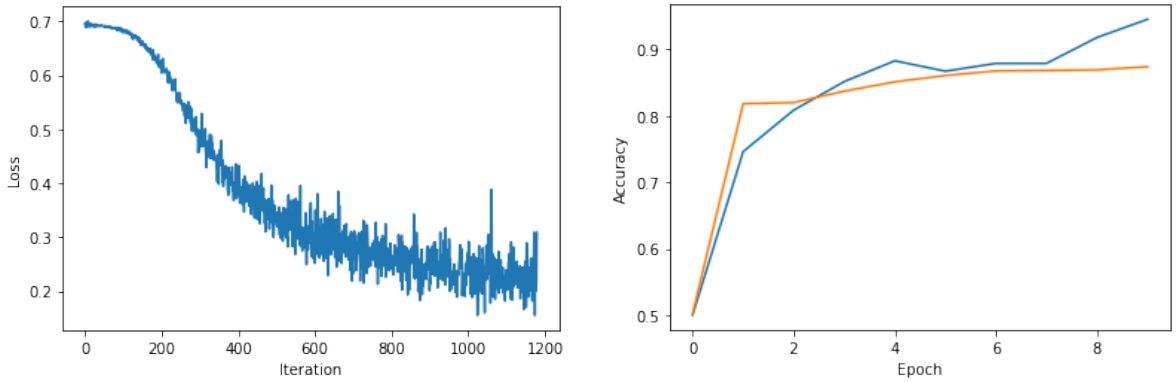

图 8: TF 在 Adam 下的 loss 和准确率

可以发现使用 SGD 根本训练不起来，使用 Adam 在训练集上准确率达到 0.9413, 在验证集上准确率达到 0.8762，最后在测试集上的准确率可以达到 **0.8778**。图9是 TF 使用 Adam 在

测试集的 precision、recall、f1-score 和 accuracy。

	precision	recall	f1-score	support
1	0.85	0.90	0.87	4751
0	0.90	0.86	0.88	5249
accuracy			0.88	10000
macro avg	0.88	0.88	0.88	10000
weighted avg	0.88	0.88	0.88	10000

图 9: TF 在 Adam 下的 precision、recall、f1-score 和 accuracy

2. TF-IDF

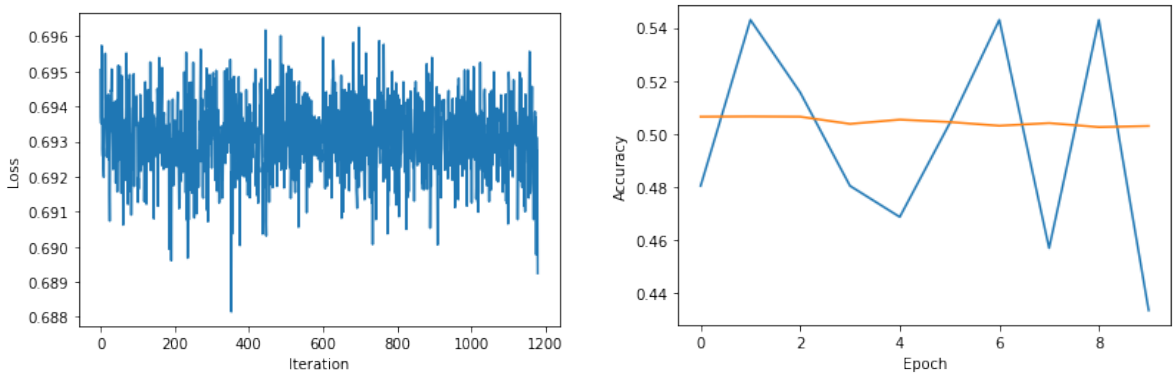

图 10: TF-IDF 在 SGD 下的 loss 和准确率

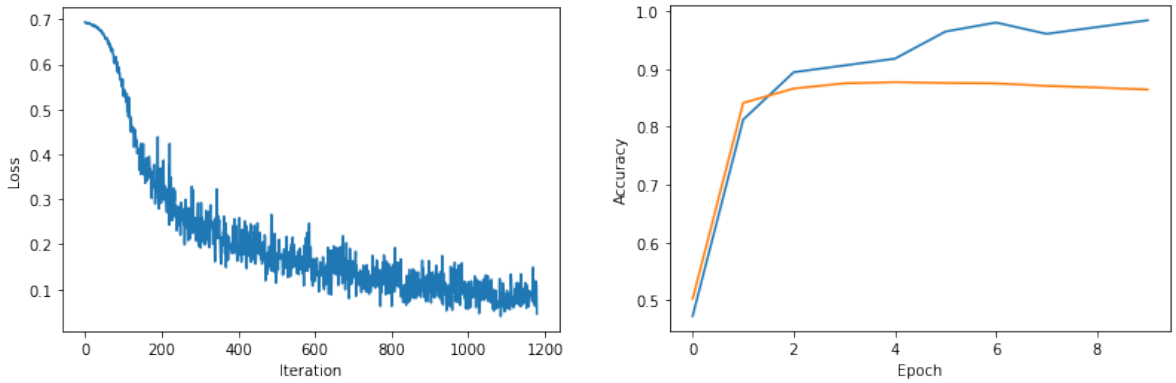

图 11: TF-IDF 在 Adam 下的 loss 和准确率

TF-IDF 使用 SGD 依然训练不起来，说明本次的分类任务 SGD 确实不行，在后面分析中都会使用 Adam。使用 Adam 在训练集上准确率达到 0.9905, 在验证集上准确率达到 0.8777, 最后在测试集上的准确率可以达到 **0.8858**。使用 TF-IDF 相比于 TF，训练集、验证集、测试集的准确率都有所提高。图12是 TF-IDF 使用 Adam 在测试集的 precision、recall、f1-score 和 accuracy。

	precision	recall	f1-score	support
1	0.88	0.88	0.88	4993
0	0.88	0.88	0.88	5007
accuracy			0.88	10000
macro avg	0.88	0.88	0.88	10000
weighted avg	0.88	0.88	0.88	10000

图 12: TF-IDF 在 Adam 下的 precision、recall、f1-score 和 accuracy

3. Word2Vec

(i) 词向量取平均

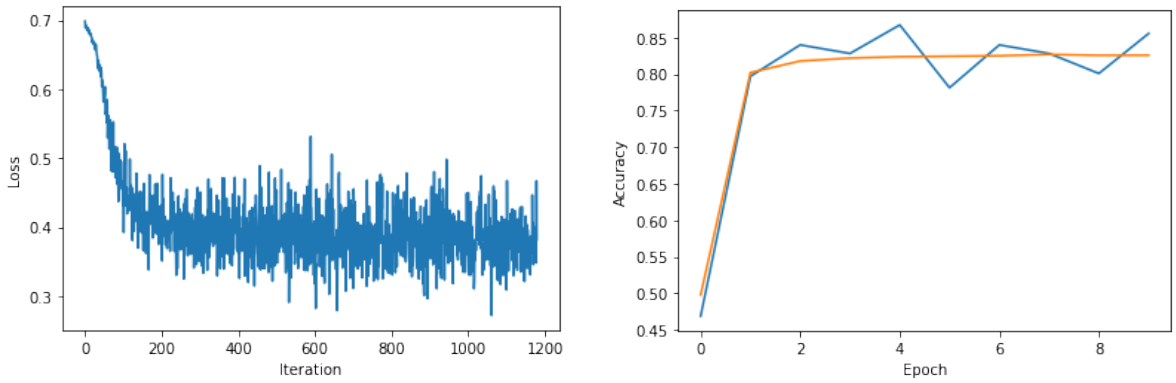

图 13: Word2Vec 词向量取平均的 loss 和准确率

Word2Vec 词向量取平均的方法效果不是很好，loss 不停震荡一直降不下去。最终在训练集上准确率达到 0.8672, 在验证集上准确率达到 0.8268，在测试集上的准确率可以达到 **0.8323**。说明 Word2Vec 词向量取平均的方法确实不是一种很好的文档表示方法。图14是 Word2Vec 词向量取平均在测试集的 precision、recall、f1-score 和 accuracy。

	precision	recall	f1-score	support
1	0.80	0.85	0.83	4681
0	0.86	0.81	0.84	5319
accuracy			0.83	10000
macro avg	0.83	0.83	0.83	10000
weighted avg	0.83	0.83	0.83	10000

图 14: Word2Vec 词向量取平均的 precision、recall、f1-score 和 accuracy

(ii) 词向量 Embedding

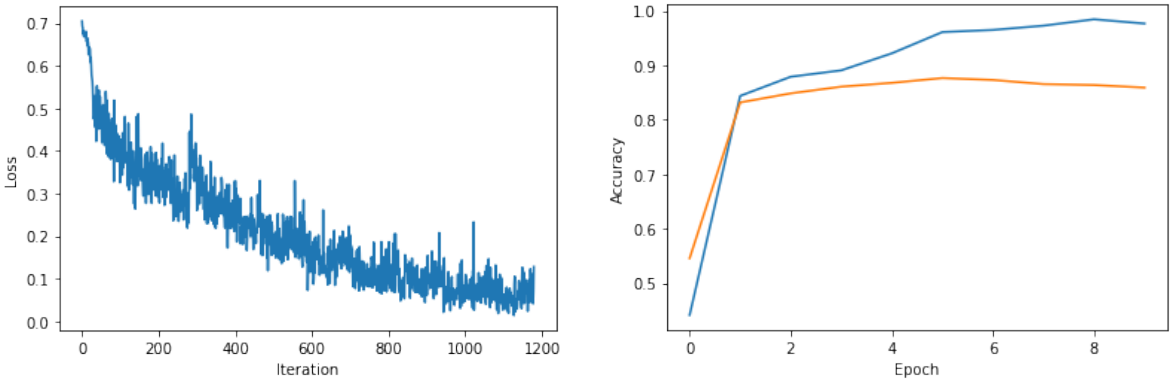

图 15: Word2Vec 词向量 Embedding 的 loss 和准确率

Word2Vec 词向量 Embedding 的方法效果对比取平均的方法就好了很多。最终在训练集上准确率达到 0.9922, 在验证集上准确率达到 0.8773，在测试集上的准确率可以达到 **0.8828**。与 TF-IDF 的结果相差不大。图16是 Word2Vec 词向量 Embedding 在测试集的 precision、recall、f1-score 和 accuracy。

	precision	recall	f1-score	support
1	0.86	0.88	0.87	4896
0	0.88	0.87	0.88	5104
accuracy			0.87	10000
macro avg	0.87	0.87	0.87	10000
weighted avg	0.87	0.87	0.87	10000

图 16: Word2Vec 词向量 Embedding 的 precision、recall、f1-score 和 accuracy

(iii) 词向量 Embedding+Attention 机制

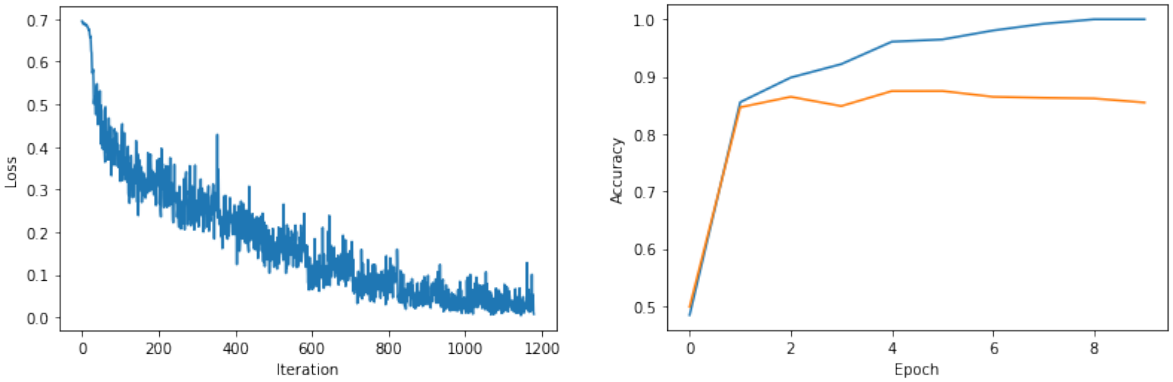

图 17: Word2Vec 词向量 Embedding+Attention 机制的 loss 和准确率

在 Word2Vec 词向量 Embedding 的基础上加入 Attention 机制，在训练集上准确率达到 1.0000, 在验证集上准确率达到 0.8742，在测试集上的准确率可以达到 **0.8759**。相比于前一种方法，加入 Attention 机制虽然在训练上已经完全正确，但在测试集上准确率并没有提高，甚至略有下降，早已过拟合。图18是 Word2Vec 词向量 Embedding+Attention 机制在测试集的 precision、recall、f1-score 和 accuracy。

	precision	recall	f1-score	support
1	0.85	0.89	0.87	4767
0	0.90	0.86	0.88	5233
accuracy			0.87	10000
macro avg	0.87	0.87	0.87	10000
weighted avg	0.87	0.87	0.87	10000

图 18: Word2Vec 词向量 Embedding+Attention 机制的 precision、recall、f1-score 和 accuracy

4. 所有方法的对比总结

下表是尝试的所有方法的准确率比较。

方法	训练集	验证集	测试集
TF	0.9413	0.8762	0.8778
TF-IDF	0.9905	**0.8777**	**0.8858**
Word2Vec 词向量取平均	0.8672	0.8268	0.8323
Word2Vec 词向量 Embedding	0.9922	0.8773	0.8828
Word2Vec 词向量 Embedding+Attention 机制	**1.0000**	0.8742	0.8759

实验发现，除了 Word2Vec 词向量取平均的方法效果较差，其他方法的结果都差不多，TF-IDF 在测试集的准确率相比其他方法稍高一些。另外，因为本次的文本二分类问题正类和负类的数量是一样多的，所以使用 precision、recall、f1-score 与准确率的结果相差不大。然而其实在本次项目中，我在训练集上的准确率已经接近甚至达到了 100%，而想要进一步解决过拟合问题，提高在测试集的准确率，可能就需要增加训练样本了。比如可以使用 **AutoEncoder 编码无标签文本**的方式，进一步增强模型的泛化能力。

七、心得体会

本次期中项目的内容是文本的二分类问题，相对来说还是比较简单的，也让我在文本处理、文本的表示方法以及深度学习的模型等方面学到很多。

其实我在本学期的《人工智能》课程中，期中项目的内容就是“幽默文本的分类和回归”，所以在这次项目上我也熟练许多。不过《人工智能》期中项目的数据集只有 8000 个样本，且不能使用 Pytorch、Tensorflow 等框架，使用的主要还是一些传统机器学习的模型，所以本次项目也让我体会到了使用深度学习框架自己“搭积木”的快乐，也对 NLP 领域产生了浓厚的兴趣。

不过也不得不感慨，基于深度学习的 NLP 训练的时间和资源都是很大的。我开始在自己笔记本的 CPU 上跑模型，结果直接卡死。还好使用了实验室的 GTX TITAN X，训练时间就大大减少了。

总而言之，这次期中项目还是十分有趣的。近几年来，Transformer、BERT 等模型的兴起给 NLP 领域又带来了新的发展空间，希望期末项目时能让我们自己来体会一下这些模型！

参考文献

[1] Word2vec.https://radimrehurek.com/gensim/models/word2vec.html,2021.

[2] pytorch loss function 总结.https://blog.csdn.net/zhangxb35/article/details/72464152?utm_source=itdadao&utm_medium=referral,2017.

[3] BiLSTM+Attention.https://blog.csdn.net/weixin_42197396/article/details/105070255,2020.

[4] 文本预处理常用技术介绍.https://blog.csdn.net/lt326030434/article/details/85240591,2018.