19335030_陈至雪 _MidtermProject

实验概述：

对IMDB进行文本分类。

数据集简介：

1、IMDB 数据集包含了50000条电影评论和它们对应的情感极性标签，可以建模为一个文本二分类问题。

分类模型：

1、三类特征：分别尝试词频 TF、TF-IDF、word2vec特征

2、一种分类方法：前馈神经网络、卷积神经网络或循环神经网络

这里采用卷积神经网络进行实现。

实验步骤：

数据处理：

读取数据：

```
#读取数据
data = pd.read_csv('C:\\Users\\azhi\\Desktop\\作业\\自然语言处理\\IMDB Dataset.csv')
review = data["review"]
sentiment = data["sentiment"]
data = np.array(data)
review = np.array(review)
sentiment = np.array(sentiment)
```

从csv文件中读取源数据，将影评保存在review中，将标签保存在sentiment中。

自定义停用词和标点：

```
#自定义停用词和标点
stopwords = ['those', 'on', 'own', ''ve', 'yourselves', 'around', 'between',
'four', 'been', 'alone', 'off', 'am', 'then', 'other', 'can', 'regarding',
'hereafter', 'front', 'too', 'used', 'wherein', ''ll', 'doing', 'everything',
'up', 'onto', 'never', 'either', 'how', 'before', 'anyway', 'since',
'through', 'amount', 'now', 'he', 'was', 'have', 'into', 'because', 'not',
'therefore', 'they', 'n't', 'even', 'whom', 'it', 'see', 'somewhere',
'thereupon', 'nothing', 'whereas', 'much', 'whenever', 'seem', 'until',
'whereby', 'at', 'also', 'some', 'last', 'than', 'get', 'already', 'our',
'once', 'will', 'noone', "'m", 'that', 'what', 'thus', 'no', 'myself', 'out',
'next', 'whatever', 'although', 'though', 'which', 'would', 'therein', 'nor',
'somehow', 'whereupon', 'besides', 'whoever', 'ourselves', 'few', 'did',
'without', 'third', 'anything', 'twelve', 'against', 'while', 'twenty', 'if',
'however', 'herself', 'when', 'may', 'ours', 'six', 'done', 'seems', 'else',
'call', 'perhaps', 'had', 'nevertheless', 'where', 'otherwise', 'still',
'within', 'its', 'for', 'together', 'elsewhere', 'throughout', 'of',
'others', 'show', ''s', 'anywhere', 'anyhow', 'as', 'are', 'the', 'hence',
'something', 'hereby', 'nowhere', 'latterly', 'say', 'does', 'neither',
'his', 'go', 'forty', 'put', 'their', 'by', 'namely', 'could', 'five',
'unless', 'itself', 'is', 'nine', 'whereafter', 'down', 'bottom', 'thereby',
'such', 'both', 'she', 'become', 'whole', 'who', 'yourself', 'every', 'thru',
'except', 'very', 'several', 'among', 'being', 'be', 'mine', 'further',
'n't', 'here', 'during', 'why', 'with', 'just', "'s", 'becomes', ''ll',
'about', 'a', 'using', 'seeming', "'d", "'ll", "'re", 'due', 'wherever',
'beforehand', 'fifty', 'becoming', 'might', 'amongst', 'my', 'empty',
'thence', 'thereafter', 'almost', 'least', 'someone', 'often', 'from',
'keep', 'him', 'or', ''m', 'top', 'her', 'nobody', 'sometime', 'across',
''s', ''re', 'hundred', 'only', 'via', 'name', 'eight', 'three', 'back',
'to', 'all', 'became', 'move', 'me', 'we', 'formerly', 'so', 'i', 'whence',
'under', 'always', 'himself', 'in', 'herein', 'more', 'after', 'themselves',
'you', 'above', 'sixty', 'them', 'your', 'made', 'indeed', 'most',
'everywhere', 'fifteen', 'but', 'must', 'along', 'beside', 'hers', 'side',
'former', 'anyone', 'full', 'has', 'yours', 'whose', 'behind', 'please',
'ten', 'seemed', 'sometimes', 'should', 'over', 'take', 'each', 'same',
'rather', 'really', 'latter', 'and', 'ca', 'hereupon', 'part', 'per',
'eleven', 'ever', ''re', 'enough', "n't", 'again', ''d', 'us', 'yet',
'moreover', 'mostly', 'one', 'meanwhile', 'whither', 'there', 'toward', ''m',
"'ve", ''d', 'give', 'do', 'an', 'quite', 'these', 'everyone', 'towards',
'this', 'cannot', 'afterwards', 'beyond', 'make', 'were', 'whether', 'well',
'another', 'below', 'first', 'upon', 'any', 'none', 'many', 'serious',
'various', 're', 'two', 'less', ''ve']
punctuation = r"""!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"""
```

对数据做清洗并构建词库：

```
#数据清洗
#去除html标记,将大写字母全部转为小写
for i in range(len(review)):
    review[i] = re.sub(r'<.*?>','',review[i])
    review[i] = review[i].lower()
    #删除标点符号
    for j in punctuation:
        review[i] = review[i].replace(j, '')
#将review从array转化成list
review = review.tolist()
sentiment = sentiment.tolist()
```

去除字符串中的html标记和标点符号，将所有字母转化为小写字母。并将review和sentiment从np.array格式转化为list格式。

构建词库：

```
#词库构建
def buildLib(corpus):
    weight_long = [eve.split() for eve in corpus]
    word_all = []
    for eve in weight_long:
        for x in eve:
            #删除停用词
            if(len(x) > 1) and (x not in stopwords) :
                word_all.append(x)
    word_counts = collections.Counter(word_all)
    word_sorted = [(l,k) for k,l in sorted([(j,i) for i,j in word_counts.items()], reverse=True)]
    #过滤低频词（出现频次小于35）
    delete = []
    for i in range(len(word_sorted)):
        (l,k) = word_sorted[i]
        if(k <= 300):
            delete.append(l)

    #去除相同的词
    word_all = list(set(word_all))
    #去除低频词
    for w  in delete:
        word_all.remove(w)
    return word_all
```

在词库构建的过程中删除了停用词，然后进行了词频统计，删除5000个影评中出现频次小于300的低频词。为了避免每次电脑关机都要重新建词库，我把构建的词库保存在了`wordLib1.txt`文件中。

```
#将词库结果写入txt文件
f=open("C:\\Users\\azhi\\Desktop\\作业\\自然语言处理\\wordLib1.txt","w",encoding='utf-8')
for word in word_all:
    f.write(word + ' ')
f.close()
```

查看词库中的词语个数：

```
print(len(word_all))
```

```
2659
```

可以看到，经过处理，词库中剩下2659个词。

处理每条数据：

为了统一影评中的词汇与词库中的词汇相同，只对影评中在词库中出现的词进行保存，并将处理过的影评重新保存在`review.txt`文件中，避免每次重新开机都要运行一遍。

```
f = open("C:\\Users\\azhi\\Desktop\\作业\\自然语言处理\\review3.txt","w",encoding='utf-8')
for i in range(len(review)):
    words = review[i].split()
    str = ''
    for word in words:
        if word in word_all:
            str += word + ' '
    f.write(str + "\n")
f.close()
```

TF+CNN：

读取数据：

```
#读取数据
with open("C:\\Users\\azhi\\Desktop\\作业\\自然语言处理\\wordLib3.txt","r",encoding='utf-8') as f:
    file = f.read()
word_all = file.split()
with open("C:\\Users\\azhi\\Desktop\\作业\\自然语言处理\\review3.txt","r",encoding='utf-8') as F:
    review = F.readlines()
f.close()
F.close()

data = pd.read_csv('C:\\Users\\azhi\\Desktop\\作业\\自然语言处理\\IMDB Dataset.csv')
Y = data["sentiment"]
Y = np.array(Y)
Y = Y.tolist()
for i in range(len(Y)):
    if Y[i] == 'positive':
        Y[i] = 1
    else:
        Y[i] = 0
```

从词库文件`wordLib3.txt`中读取词库，保存在word_all列表中；从`review3.txt`文件中读取处理好的文本，保存在review

列表中。从原始数据中读取标签列表，保存在Y中，并将“positive”转化为1，将“negative”转化为0。

计算TF：

TF是词频，表示一个词在文章中出现的频次。这个数字通常会被归一化(一般是词频除以文章总词数), 以防止它偏向长的文件。计算公式如下：

$$TF_w = \frac{\text{在某一类中词条}w\text{出现的次数}}{\text{该类中所有的词条数目}} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

```
#计算TF
def TF(corpus,word_all):
    weight_tf = [[] for i in corpus]
    for word in word_all:
        for i in range(len(corpus)):
            temp_list = corpus[i].split()
            n1 = temp_list.count(word)
            n2 = len(temp_list)
            tf = n1/n2
            weight_tf[i].append(tf)
    return weight_tf
```

传入的参数中corpus是含有50000个影评的list，word_all是构建的词库，返回计算好的tf列表，列表长度为50000，一个元素代表一篇影评的tf特征，每个tf长为2659——词汇表的长度。

然后计算tf，并将tf写入文件：

```
tf = TF(review,word_all)
#将tf写入txt文件
f=open("C:\\Users\\azhi\\Desktop\\作业\\自然语言处理
\\tf3.txt","w",encoding='utf-8')
for vec in tf:
    for num in vec:
        f.write(str(num) + ' ')
    f.write('\n')
f.close()
```

下次使用tf时，可从文件中直接读取：

```
#从文件中读tf
f=open("C:\\Users\\azhi\\Desktop\\作业\\自然语言处理
\\tf3.txt","r",encoding='utf-8')
TF = f.readlines()
f.close()
tf = [[]for i in range(len(TF))]
for i in range(len(TF)):
    for num in TF[i].split():
        tf[i].append(float(num))
```

划分训练集、验证集和测试集：

```
tf_train = tf[0:30000]
tf_valid = tf[30000:40000]
tf_test = tf[40000:]
y_train = Y[0:30000]
y_valid = Y[30000:40000]
y_test = Y[40000:]
```

手动划分训练集为前30000条影评，验证集为第30001~40000条影评，测试集为第40001~50000条影评。

构建神经网络模型：

```
model = Sequential()
model.add(Conv1D(filters = 64, kernel_size = 3,input_shape=(2659, 1), padding = 'same', activation = 'relu'))
model.add(MaxPooling1D(pool_size = 2))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(64, activation = 'relu'))
model.add(Dense(1, activation = 'sigmoid'))
model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])
print(model.summary())
```

输入是 2659 * 1 的向量，下面通过 1 层卷积层与池化层来缩小向量长度，再加 1 层 Flatten 层将向量压缩到 1 维，最后通过 2 层 Dense（全连接层）将向量长度收缩到 1 。

由于 IMDB 情感数据集只有正负两个类别，因此全连接层是只有一个神经元的二元分类，使用 `sigmoid` 激活函数。

在该样例的二元分类器中，使用了二元交叉熵作为损失函数，使用 `adam` 作为优化器，使用 `accuracy` 作为评估矩阵。

神经网络结构如下：

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d_11 (Conv1D)          (None, 2659, 64)          256

 max_pooling1d_11 (MaxPoolin  (None, 1329, 64)         0
 g1D)

 dropout_11 (Dropout)        (None, 1329, 64)          0

 flatten_6 (Flatten)         (None, 85056)             0

 dense_17 (Dense)            (None, 64)                5443648

 dense_18 (Dense)            (None, 1)                 65

=================================================================
Total params: 5,443,969
Trainable params: 5,443,969
Non-trainable params: 0
_________________________________________________________________
None
```

训练神经网络：

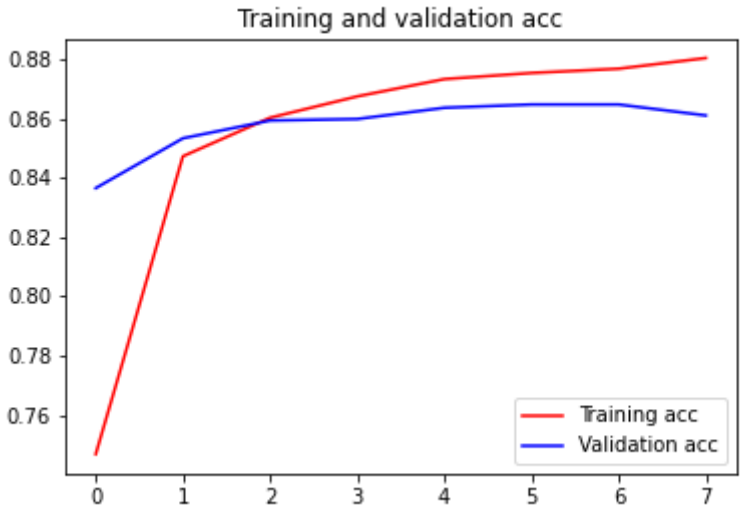
```
# 训练
history = model.fit(tf_train, y_train, epochs=8, batch_size=128,
                    verbose=1, validation_data=[tf_valid, y_valid])
model_file = 'C:\\Users\\azhi\\Desktop\\作业\\自然语言处理\\tf_model.h5'
model.save(model_file)
```

将训练好的神经网络模型保存为 `tf_model.h5`。

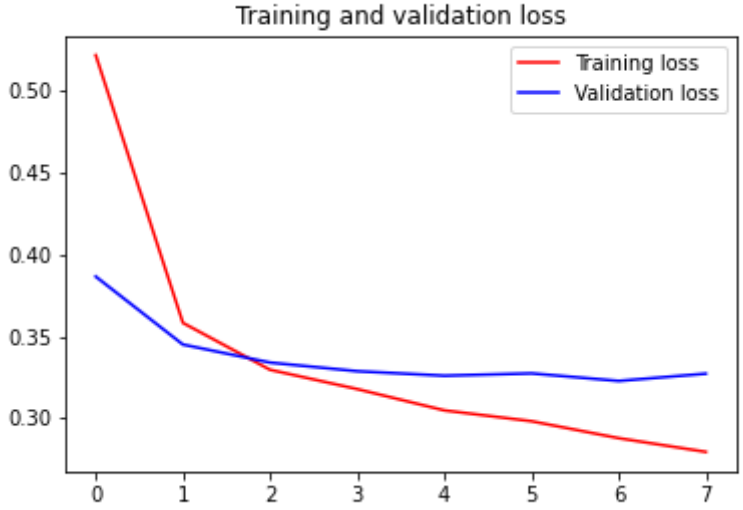
绘制acc和loss曲线：

```
plt.figure()
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(len(loss))
plt.plot(epochs,acc,'r',label = 'Training acc')
plt.plot(epochs,val_acc,'b',label = 'Validation acc')
plt.title('Training and validation acc')
plt.legend()
plt.savefig('C:\\Users\\azhi\\Desktop\\作业\\自然语言处理\\tf_acc8.png')
plt.show()

plt.plot(epochs,loss,'r',label = 'Training loss')
plt.plot(epochs,val_loss,'b',label = 'Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.savefig('C:\\Users\\azhi\\Desktop\\作业\\自然语言处理\\tf_loss8.png')
plt.show()
```



可以看到训练过程中，训练集和验证集的准确率都在上升，说明模型训练良好。



可以看到loss在训练集和验证集上的表现都呈现下降趋势，说明模型训练良好。

测试神经网络并计算精度：

```
tf_pred = model.predict(tf_test)
for i in range(len(tf_pred)):
    if(tf_pred[i] > 0.5):
        tf_pred[i] = 1
    else:
        tf_pred[i] = 0

acc = cal_acc(tf_pred,y_test)
TP, FN, FP, TN = cal_TP_FN_FP_TN(tf_pred,y_test)
P = cal_P(TP,FP)
R = cal_R(TP,FN)
print("acc is "+str(acc[0]))
print("P is "+str(P))
print("R is "+str(R))
```

因为该问题是一个二分类问题，故对预测出来的tf_pred值进行简单的处理，若tf_pred的值小于0.5，则将其归为0类，否则将其归为1类。

TF-IDF+CNN:

先读取数据，实现和tf的一样。

计算TF-IDF:

IDF是逆向文件频率，某一特定词语的IDF，可以由总文件数目除以包含该词语的文件的数目，再将得到的商取对数得到。计算公式为：

$$idf_i = log(\frac{\text{语料库的文档总数}+1}{\text{包含词条}w\text{的文档数}+1}) = log\frac{|D|+1}{|\{j:t_i \in d_j\}+1|}$$

TF-IDF的主要思想是：如果某个词或短语在一篇文章中出现的频率TF高，并且在其他文章中很少出现，则认为此词或者短语具有很好的类别区分能力，适合用来分类。TFIDF实际上是：TF-IDF.

TF-IDF的实现如下：

```
def IDF(corpus, word_dict):
    n3 = len(corpus)
    idf = []
    for word in word_dict.keys():
        #print(word)
        n4 = 0
        for j in range(len(corpus)):
            if word in corpus[j].split():
                n4 += 1
        idf.append(math.log(((n3+1)/(n4+1)),10))
    return idf

idf = IDF(review,word_index)

#把idf写如文件
```

```
f=open("C:\\Users\\azhi\\Desktop\\作业\\自然语言处理
\\idf3.txt","w",encoding='utf-8')
for num in idf:
    f.write(str(num) + ' ')
f.write('\n')
f.close()
```

IDF函数计算每个词的IDF，并将其保存在一个列表中。这里将idf结果保存在文件`idf3.txt`中。

然后利用idf列表计算TF_IDF，并把 tfidf 保存在`tfidf3.txt`文件中：

```
#计算TF-IDF并返回
def TF_IDF(corpus, word_dict, idf):
    weight = [[] for i in corpus]
    for word in word_dict.keys():
        for i in range(len(corpus)):
            temp_list = corpus[i].split()
            n1 = temp_list.count(word)
            n2 = len(temp_list)
            tf = n1/n2
            index = word_dict[word]-1
            weight[i].append(tf*idf[index])
    #l2_weight = L2(weight)
    return weight

tfidf = TF_IDF(review,word_index,idf)

#把tfidf写如文件
f=open("C:\\Users\\azhi\\Desktop\\作业\\自然语言处理
\\tfidf3.txt","w",encoding='utf-8')
for vec in tfidf:
    for num in vec:
        f.write(str(num) + ' ')
    f.write('\n')
f.close()
```

划分训练集、验证集和测试集：

```
tfidf_train = tfidf[0:30000]
tfidf_valid = tfidf[30000:40000]
tfidf_test = tfidf[40000:]
y_train = Y[0:30000]
y_valid = Y[30000:40000]
y_test = Y[40000:]
```

手动划分训练集为前30000条影评，验证集为第30001~40000条影评，测试集为第40001~50000条影评。

构建神经网络：

```
model = Sequential()
model.add(Conv1D(filters = 64, kernel_size = 3,input_shape=(2659, 1), padding = 'same', activation = 'relu'))
model.add(MaxPooling1D(pool_size = 2))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(64, activation = 'relu'))
model.add(Dense(1, activation = 'sigmoid'))
model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])
print(model.summary())
```

输入是 2659 * 1 的向量，下面通过 1 层卷积层与池化层来缩小向量长度，再加一层 Flatten 层将向量压缩到 1 维，最后通过 2 层 Dense（全连接层）将向量长度收缩到 1 。

由于 IMDB 情感数据集只有正负两个类别，因此全连接层是只有一个神经元的二元分类，使用 `sigmoid` 激活函数。

在该样例的二元分类器中，使用了二元交叉熵作为损失函数，使用 `adam` 作为优化器，使用 `accuracy` 作为评估矩阵。

神经网络结构如下：

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d_8 (Conv1D)           (None, 2659, 64)          256

 max_pooling1d_8 (MaxPooling  (None, 1329, 64)         0
 1D)

 dropout_8 (Dropout)         (None, 1329, 64)          0

 flatten_4 (Flatten)         (None, 85056)             0

 dense_12 (Dense)            (None, 64)                5443648

 dense_13 (Dense)            (None, 1)                 65

=================================================================
Total params: 5,443,969
Trainable params: 5,443,969
Non-trainable params: 0
_________________________________________________________________
None
```

训练神经网络：

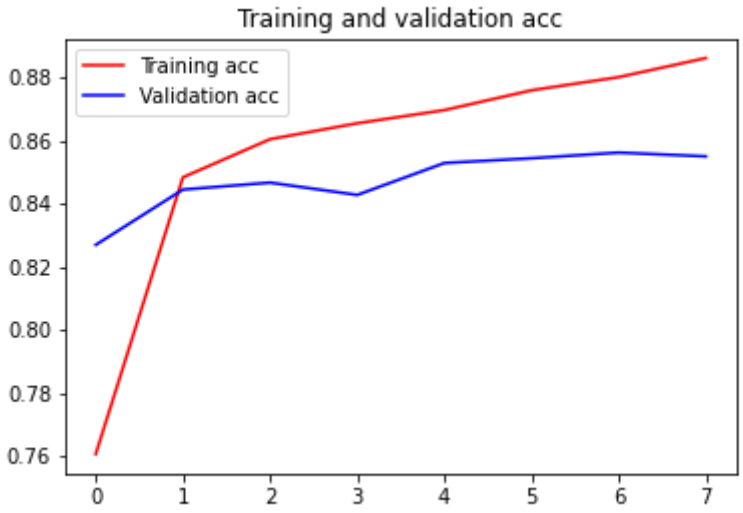
```
# 训练
history = model.fit(tfidf_train, y_train, epochs=8, batch_size=128,
                    verbose=1, validation_data=[tfidf_valid, y_valid])
model_file = 'C:\\Users\\azhi\\Desktop\\作业\\自然语言处理\\tfidf_model.h5'
model.save(model_file)
```

将训练好的神经网络模型保存为 `tfidf_model.h5`。

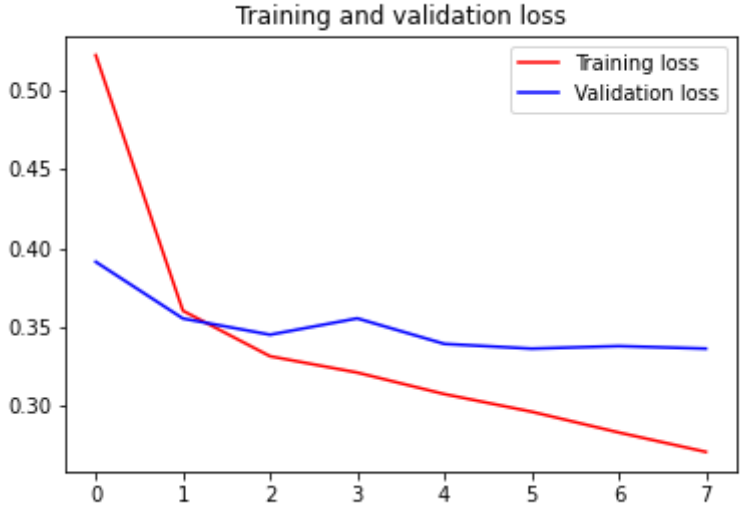
绘制acc和loss曲线：

```
plt.figure()
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(len(loss))
plt.plot(epochs,acc,'r',label = 'Training acc')
plt.plot(epochs,val_acc,'b',label = 'Validation acc')
plt.title('Training and validation acc')
plt.legend()
plt.savefig('C:\\Users\\azhi\\Desktop\\作业\\自然语言处理\\tfidf_acc8.png')
plt.show()

plt.plot(epochs,loss,'r',label = 'Training loss')
plt.plot(epochs,val_loss,'b',label = 'Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.savefig('C:\\Users\\azhi\\Desktop\\作业\\自然语言处理\\tfidf_loss8.png')
plt.show()
```



可以看到训练过程中，训练集和验证集的准确率都在上升，说明模型训练良好。



可以看到loss在训练集和验证集上的表现都呈现下降趋势，说明模型训练良好。

测试神经网络并计算精度：

```
tfidf_pred = model.predict(tfidf_test)
for i in range(len(tfidf_pred)):
    if(tfidf_pred[i] > 0.5):
        tfidf_pred[i] = 1
    else:
        tfidf_pred[i] = 0

acc = cal_acc(tfidf_pred,y_test)
TP, FN, FP, TN = cal_TP_FN_FP_TN(tfidf_pred,y_test)
P = cal_P(TP,FP)
R = cal_R(TP,FN)
print("acc is "+str(acc[0]))
print("P is "+str(P))
print("R is "+str(R))
```

因为该问题是一个二分类问题，故对预测出来的tf_pred值进行简单的处理，若tf_pred的值小于0.5，则将其归为0类，否则将其归为1类。

其中计算精度acc、查准率P和查全率R的函数实现和tf模块中的一样。

Word2vec+CNN:

读取数据和前面步骤一样。

训练word2vec模型：

```
sentence = []
for i in range(len(review)):
    w = review[i].split()
    sentence.append(w)
```

先将每个元素为字符串（一个字符串为一个影评）的列表review转化为每个元素为字符串列表的列表sentence。

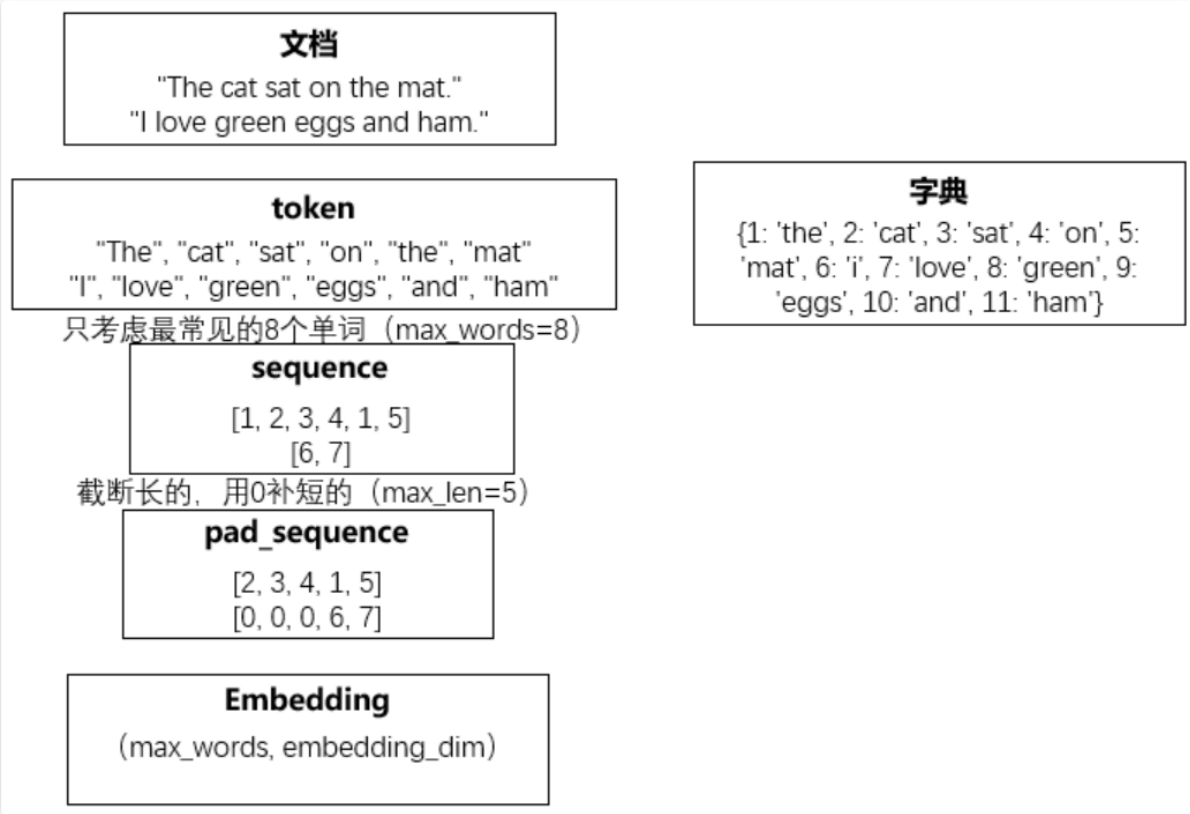
训练word2vec模型，并将模型保存为`word2vec3.bin`：

```
model=gensim.models.Word2Vec(sentence,sg=1,vector_size=100,window=5,min_count=2,negative=3,sample=0.001,hs=1,workers=4)
model.wv.save_word2vec_format("C:\\Users\\azhi\\Desktop\\作业\\自然语言处理\\word2vec3.bin",binary = "Ture")
```

输入的预料是之前处理好的sentence列表。sg=1，采用skip-gram算法。规定vector_size为100，即每个词向量的维度为100。min_count设置为2，词频少于2的单词会被丢弃掉，但由于之前对数据进行过低频词的过滤，所以这里不设置也无所谓。negative = 3，设置3个noise words。 workers=4，控制训练的并行数为4。设置hs=1，采用hierarchica·softmax技巧。

Embedding:

过程描述如下：

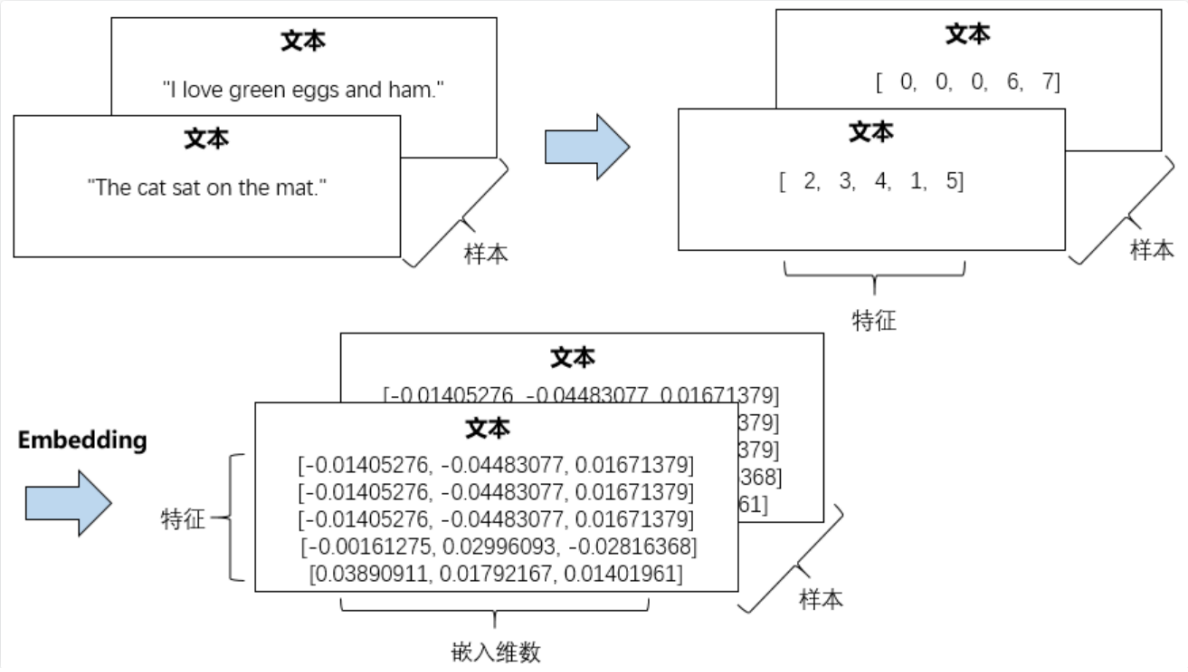


Keras 的 **Embedding 层的输入是一个二维整数张量**， 形状为（samples，sequence_length），即（样本数，序列长度）

较短的序列应该用 0 填充，较长的序列应该被截断，保证输入的序列长度是相同的。

Embedding 层输出是（samples，sequence_length，embedding_dimensionality） 的**三维浮点数张量。**

从样本的角度描述如下：



对文本进行分词处理：

```
tokenizer = Tokenizer()
tokenizer.fit_on_texts(review)
```

对分词结果进行序列化：

```
sequences = tokenizer.texts_to_sequences(review)
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

统一输入的序列长度：

```
data = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)
```

其中`MAX_SEQUENCE_LENGTH = 100`，规定每个文本的词数不超过100（经统计，平均每个影评的词数为70，因此保留为100是合理的），若超过则进行截断，若不足100，则进行补0。

构造embedding_matrix：

```
# 加载bin格式的模型
word2vec =
gensim.models.KeyedVectors.load_word2vec_format("word2Vec3.bin",binary=True)
embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM))
for word, i in word_index.items():
    if str(word) in word2vec:
        embedding_matrix[i] = np.asarray(word2vec[str(word)],
                                         dtype='float32')
```

embedding_matrix是有词库中所有词的词向量组成的矩阵。先加载之前训练好的word2vec模型，然后通过该模型得到每个词的词向量，放入embedding_matrix。

定义embedding层：

```
embedding_layer = Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=False)
```

embedding层的作用是将文本处理成向量。使用预训练的 word2vec 代替 embedding 层，需要将 embedding 层的参数用 word2vec 模型中的词向量替换。替换后的 embedding 矩阵形状为 2660 x 100， 2660 行代表 2660 个单词，每一行的这长度 100 的行向量对应这个词在 word2vec 空间中的 100 维向量。最后，设定 embedding 层的参数固定，不参加训练，这样就把预训练的 word2vec 嵌入到了深度学习的模型之中。EMBEDDING_DIM = 100，为词向量的维度。最后把统一长度的序列化结果输入到 Embedding 层中，embedding 层将输出一个 MAX_SEQUENCE_LENGTH * EMBEDDING_DIM 的矩阵。

划分训练集、验证集和测试集：

```
w2v_train = data[0:30000]
w2v_valid = data[30000:40000]
w2v_test = data[40000:]
y_train = Y[0:30000]
y_valid = Y[30000:40000]
y_test = Y[40000:]
y_train = y_train.astype(np.float32)
y_test = y_test.astype(np.float32)
y_valid = y_valid.astype(np.float32)
```

手动划分训练集为前30000条影评的序列，验证集为第30001~40000条影评序列，测试集为第40001~50000条影评序列。

构建神经网络：

```
#word2vec
w2vmodel = Sequential()
w2vmodel.add(embedding_layer)
w2vmodel.add(Dropout(0.2))
w2vmodel.add(Conv1D(filters = 250, kernel_size = 3, padding = 'same',
activation = 'relu'))
w2vmodel.add(MaxPooling1D(pool_size = 2))
w2vmodel.add(Dropout(0.5))
w2vmodel.add(Conv1D(filters = 128, kernel_size = 3, padding =
'same',activation = 'relu'))
w2vmodel.add(MaxPooling1D(pool_size = 2))
w2vmodel.add(Dropout(0.5))
w2vmodel.add(Flatten())
w2vmodel.add(Dense(125, activation = 'relu'))
w2vmodel.add(Dense(1, activation = 'sigmoid'))
print(w2vmodel.summary())
```

首先是一个将文本处理成向量的 embedding 层，这样每个文档被处理成一个 MAX_SEQUENCE_LENGTH * EMBEDDING_DIM 的二维向量，下面通过 2 层卷积层与池化层来缩小向量长度，再加一层 Flatten 层将向量压缩到 1 维，最后通过两层 Dense（全连接层）将向量长度收缩到 1 。

由于 IMDB 情感数据集只有正负两个类别，因此全连接层是只有一个神经元的二元分类，使用 `sigmoid` 激活函数。

在该样例的二元分类器中，使用了二元交叉熵作为损失函数，使用 `adam` 作为优化器，使用 `accuracy` 作为评估矩阵。

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=
['accuracy'])
```

神经网络结构如下：

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_7 (Embedding)      (None, 100, 100)          266000

dropout_20 (Dropout)         (None, 100, 100)          0

conv1d_13 (Conv1D)           (None, 100, 250)          75250

max_pooling1d_13 (MaxPoolin  (None, 50, 250)           0
g1D)

dropout_21 (Dropout)         (None, 50, 250)           0

conv1d_14 (Conv1D)           (None, 50, 128)           96128

max_pooling1d_14 (MaxPoolin  (None, 25, 128)           0
g1D)

dropout_22 (Dropout)         (None, 25, 128)           0

flatten_7 (Flatten)          (None, 3200)              0

dense_15 (Dense)             (None, 125)               400125

dense_16 (Dense)             (None, 1)                 126

=================================================================
Total params: 837,629
Trainable params: 571,629
Non-trainable params: 266,000
_________________________________________________________________
None
```

训练神经网络：

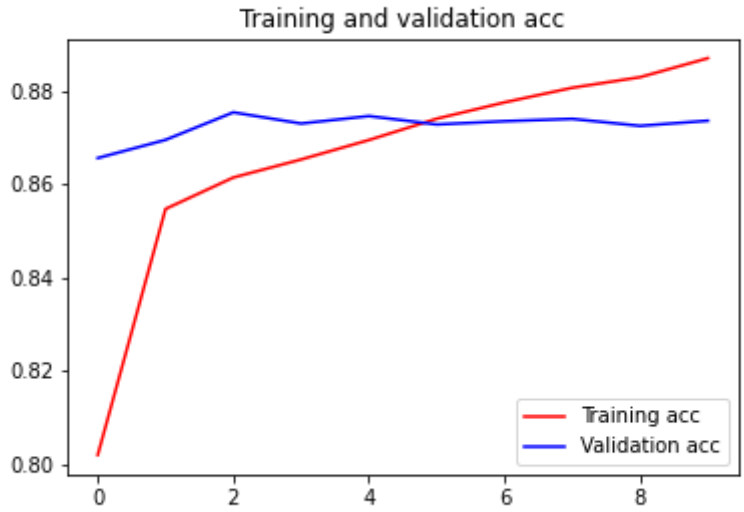
```
hist = w2vmodel.fit(w2v_train, y_train, validation_data=(w2v_valid, y_valid),
epochs=10, batch_size=100)
w2vmodel.save('word_vector_cnn.h5')
```

将神经网络模型保存为`word_vector_cnn.h5`.

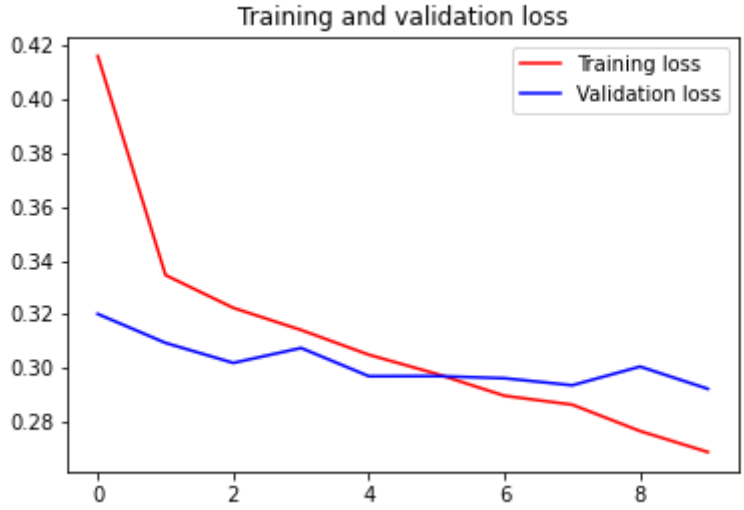
绘制acc和loss曲线：

```
plt.figure()
acc = hist.history['accuracy']
val_acc = hist.history['val_accuracy']
loss = hist.history['loss']
val_loss = hist.history['val_loss']
epochs = range(len(loss))
plt.plot(epochs,acc,'r',label = 'Training acc')
plt.plot(epochs,val_acc,'b',label = 'Validation acc')
plt.title('Training and validation acc')
plt.legend()
plt.show()
```

```
12
13 plt.plot(epochs,loss,'r',label = 'Training loss')
14 plt.plot(epochs,val_loss,'b',label = 'Validation loss')
15 plt.title('Training and validation loss')
16 plt.legend()
17 plt.show()
```



可以看到在Training数据集上精度越来越好，在验证集上的精度也有缓慢增长的趋势。说明模型训练良好。



可以看到loss在训练集和验证集上的表现都呈现下降趋势，说明模型训练良好。

测试神经网络并计算精度：

```
1  w2v_pred = w2vmodel.predict(w2v_test)
2  w2v_list = []
3  for i in range(len(w2v_pred)):
4      if(w2v_pred[i][0] < 0.5):
5          w2v_list.append(0)
6      else:
7          w2v_list.append(1)
8
9  w2vacc = cal_acc(w2v_list,y_test)
10 w2vTP, w2vFN, w2vFP, w2vTN = cal_TP_FN_FP_TN(w2v_list,y_test)
```

```
w2vP = cal_P(w2vTP,w2vFP)
w2vR = cal_R(w2vTP,w2vFN)

print("acc is "+str(w2vacc))
print("P is "+str(w2vP))
print("R is "+str(w2vR))
```

因为该问题是一个二分类问题，故对预测出来的tf_pred值进行简单的处理，若tf_pred的值小于0.5，则将其归为0类，否则将其归为1类。

其中计算精度acc、查准率P和查全率R的函数实现和tf模块中的一样。

结果分析：

使用准确率、精度、召回率三大指标评价模型训练的好坏。

准确率（Accuracy）：

准确率（Accuracy）描述的是模型预测正确（包括真正例、正负例）的样本数量站总样本的比例，即模型预测的准确率，计算公式如下所示：

$$Accuracy = \frac{预测正确的数目}{总的预测数目} = 1 - \frac{预测错误的数目}{总的预测数目}$$

实现如下：

```
def cal_acc(pred,y_test):
    acc = 0;
    for i in range(len(pred)):
        acc += abs(pred[i]-y_test[i])
    acc /= len(pred)
    acc = 1-acc
    return acc
```

由于处理后的 pred 要么为1，要么为0， 因此要是预测的 pred 值与真实值不一致，则它们的差的绝对值为1，否则为 0，将测试集的所有pred 值与真实的 label 值做差相加，即为预测错误的数目，计算出错误率，然后再计算出准确率。

精度（Precision）：

精度也叫查准率，它描述的是模型预测为True且实际为True的样本数量占模型总预测为True的样本数量的比例，计算公式如下所示：

$$Precision = \frac{真正例}{真正例 + 假正例} = \frac{TP}{TP + FP}$$

实现如下：

```
def cal_P(TP,FP):
    P = TP/(TP+FP)
    return P
```

精度反应了模型**预测的准确度**，但是精度高的模型不一定好，假设模型只预测出来一个True样本，且样本实际也是True，那么此时的Precision是1，但是模型会预测到很多的假反例，这样的模型依然是不理想的，因此仍然要引入其它评价标准，如查全率。

召回率（Recall）：

召回率也叫查全率，它描述的是模型预测为True且实际为True的样本数量占测试数据集中实际为True样本的总数的比例，计算公式如下所示：

$$Recall = \frac{\text{真正例}}{\text{真正例} + \text{假负例}} = \frac{TP}{TP + FN}$$

实现如下：

```
def cal_R(TP,FN):
    R = TP / (TP+FN)
    return R
```

召回率越高表示模型能够找到测试样本中True样本的能力越强，但是如果模型将所有的测试样本全部预测为True，此时的Recall值为1，但是这样的模型仍然不是一个理想的模型。

其中，

真正例（True Positive，TP）：模型将测试样本中True类型的样本预测为True的样本数量
假负例（False Negative，FN）：模型将测试样本中True类型的样本预测为False的样本数量
假正例（False Positive，FP）：模型将测试样本中False类型的样本预测为True的样本数量
真负例（True Negative，TN）：模型将测试样本中False类型的样本预测为False的样本数量

计算过程如下：

```
def cal_TP_FN_FP_TN(pred,y_test):
    TP = 0
    FN = 0
    FP = 0
    TN = 0
    for i in range(len(pred)):
        if(pred[i] == 1 and y_test[i] == 1):
            TP += 1
        if(pred[i] == 1 and y_test[i] == 0):
            FP += 1
        if(pred[i] == 0 and y_test[i] == 0):
            TN += 1
        if(pred[i] == 0 and y_test[i] == 1):
            FN += 1
    return TP,FN,FP,TN
```

模型结果：

模型	TF	TF-IDF	Word2vec
Accuracy	0.86	0.8543	0.8807
Precision	0.8406	0.8580	0.8901
Recall	0.8889	0.8496	0.8690

可以看到三种特征提取的方法预测的准确率都达到了85%以上，精度和召回率也有80%以上。其中word2vec的效果是最好的。

实验心得：

此次实验让我清楚了训练一个文本分类模型的过程，首先是数据预处理，如数据清洗（去停用词、去html标识符、去标点符号、去低频词等），统一数据格式，如将大写字母转化为小写字母，统一文本的长度等。

然后是选取一种能提取文本特征的方法，对文本进行特征提取，在对文本进行特征提取的过程也是将文本向量化的过程。

下一步就是选取并构建用于分类的训练模型，用之前得到的文本向量和标签训练模型。

训练一个神经网络模型通常需要把数据集划分为训练集、验证集和测试集。训练集用于训练神经网络模型，验证集用于在训练神经网络模型的过程调整模型的超参数以及用于对模型的能力进行初步评估。测试集则是用来评估模最终模型的泛化能力。

最后，评价模型的好坏，我们可能需要从一些参数出发，评判模型的表现，对于二分类问题，常用的评估参数有Accuracy（准确率）、Precision（精度）以及 Recall（召回率）。

在实验中我更能体会到课堂上学的知识有什么用，以及具体应该怎么用。在实践过程中一方面巩固了课堂上所学知识，另一方面我也认识到自己还有许多知识（如神经网络方面的知识）了解不够，有待加强。