实验报告

实验要求

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（positive or negative），将数据集建模为一个文本二分类问题

数据下载地址：https://www.kaggle.com/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews

为了方便验收，统一数据集的划分标准。具体为0-30000条为训练集、30001-40000条为验证集和40001-50000条为测试集。模型的性能以测试集的结果为最终标准

实验步骤包括下面两大步骤：

数据清洗和文本预处理

- 原始数据是爬虫得到的初步结果，里面包含了HTML标签和URL等与文本情感无关的噪音，需要进行数据清洗
- 自行决定是否要统一单词大小写、去除停用词和低频词以及标点符号等

分类模型

- 三类特征：分别尝试词频TF、TF-IDF、 word2vec特征
- 一种分类方法：前馈神经网络、卷积神经网路或循环神经网络
- 不允许使用现成的线上情感分析平台（API）

实验概述

本次实验中，将分别使用tf、tf-idf、word2vec的3种方式来获得词向量，数据处理文本预处理部分代码共用，输入同一cnn网络模型。

实验过程

数据清洗和文本预处理

对于每一条review，进行下面的操作：

去除html标签

```
pattern = re.compile(r'<[^>]+>', re.S)
review = pattern.sub('', dataset['review'][i])
```

去除标点符号（即只保留英文字母）

```
review = re.sub('[^a-zA-z]', ' ', review)
```

全部转变为小写字母

`review = review.lower()`

将sentiment表示为0/1

```
sen = dataset['sentiment'][i]
if sen == 'positive':
    sentiment.append(1)
else:
    sentiment.append(0)
```

1表示积极，0表示消极

去除高低频词

（注意，本步骤中用到的"word.txt"和"word_freq.txt"为先前专门编写代码生成的，正式的预处理中可以直接使用这两个文件）

首先，将每条review中的词语保存在`word.txt`中

```
fword = open("word.txt", 'w', encoding='utf-8')
...
    review = [word for word in review if not word in
set(stopwords.words('english'))]
    for word in review:
        fword.write(word + "\n")
...
fword.close()
```

然后，载入词语集合，去除高低频词：

```
# 载入词表
f = open('word.txt', 'r')
sourceInLines = f.readlines()  #按行读出文件内容
f.close()
words = []
for line in sourceInLines:
    temp1 = line.strip('\n')        #去掉每行最后的换行符'\n'
    words.append(temp1)

# 去除高低频词,去除后的词存于"word_freq.txt"中
Min = 1000
Max = 25000
with open("word_freq.txt", 'w', encoding='utf-8') as fout:
    for word, freq in Counter(words).most_common():
        if freq > Min and freq < Max:
            fout.write(word+"\n")
```

这里选择最低频率为1000、最高频率为25000

将去除高低频词后的词语存于`word_freq.txt`中

`word_freq.txt`中的词即为我们最终保留的词，用它来构成语料库的词典`vocab`：

```
with open("word_freq.txt", encoding='utf-8') as fin:
    vocab = [i.strip() for i in fin]
vocab = set(vocab)
```

在文本预处理中，仅保留vocab中的词（这样就起到了去除高低频词的作用）：

```
review = [word for word in review if word in vocab]
```

这里选择去除高频词，是考虑到，一般的文本处理中会去除停用词，但是对于情感分类的文本来说，去除停用词可能不是一个好的选择，因为会出现这样的情况：“isn't good”，“isn't”属于停用词，将被去除，那么整句话从消极转变为了积极。

但是仍然会有别的不属于否定词的常见词在干扰，所以我们这里选择去除高频词，就是去除掉这部分词。频率在25000以上的去除掉，这个考虑是因为25000是50000的一半，在语料库中，积极消极对半开，所以若一个词出现在了一半以上的文本里，认为它是无情感的。

完整代码

```
# 初始化vocab
with open("word_freq.txt", encoding='utf-8') as fin:
    vocab = [i.strip() for i in fin]
vocab = set(vocab)

dataset = pd.read_csv("IMDB Dataset.csv")
#数据清洗&文本预处理
corpus = []
sentiment = []
# fword = open("word.txt", 'w', encoding='utf-8')
for i in range(50000):
    pattern = re.compile(r'<[^>]+>', re.S)
    review = pattern.sub('', dataset['review'][i])
    review = re.sub('[^a-zA-z]', ' ', review)
    review = review.lower()
    review = review.split()
    # for word in review:
        # fword.write(word + "\n")
    review = [word for word in review if word in vocab]
    sen = dataset['sentiment'][i]
    if len(review)==0:
        if sen =='positive':
            review.append('wonderful')
        else:
            review.append('boring')
```

```
    review = ' '.join(review)
    corpus.append(review)
    if sen == 'positive':
        sentiment.append(1)
    else:
        sentiment.append(0)
# fword.close()
dataframe = pd.DataFrame({'sentiment': sentiment, 'review': corpus})
dataframe.to_csv("clean.csv")


# 载入预处理后的数据
train_data = pd.read_csv("clean.csv", index_col=0)

'''
# 载入词表
f = open('word.txt', 'r')
sourceInLines = f.readlines()  #按行读出文件内容
f.close()
words = []
for line in sourceInLines:
    temp1 = line.strip('\n')        #去掉每行最后的换行符'\n'
    words.append(temp1)

# 去除高低频词,去除后的词存于“word_freq.txt”中
Min = 1000
Max = 25000
with open("word_freq.txt", 'w', encoding='utf-8') as fout:
    for word, freq in Counter(words).most_common():
        if freq > Min and freq < Max:
            fout.write(word+"\n")

'''
```

corpus用于存储处理好的review

循环，对于50000条review分别进行上述操作，将处理好的review加入corpus

将corpus存储到`clean.csv`中，方便查看比对

TF

TF（Term Frequency）：中文意思是词频，也就是在一段文本中出现的频率较高的词，由于我们在之前的预处理中已经去掉了英文中的停词（类似与to，is，are，the这些高频出现但是却没有真正的实际意义的词汇）所以这里我们往往可以认为出现频率越高的词汇会对整个文档有较大的影响。

TF的计算公式如下：

TF (w,d)= count (w, d) / sum { i = 1..n | count (w, d[i]) }，就是某个词出现次数除以总词数量

获取tf的代码如下：

```
word_id = {}
```

```
vob = {}


def get_tf(corpus):
    id = 0
    for single_corpus in corpus:
        if isinstance(single_corpus, list):
            pass
        if isinstance(single_corpus, str):
            single_corpus = single_corpus.strip("\n").split(" ")
        for word in single_corpus:
            if word not in vob:
                vob[word] = 1
                word_id[word] = id
                id += 1
            else:
                vob[word] += 1

    # 生成矩阵
    X = np.zeros((len(corpus), len(vob)))
    for i in range(len(corpus)):
        if isinstance(corpus[i], str):
            single_corpus = corpus[i].strip("\n").split(" ")
        else:
            single_corpus = corpus[i]
        for j in range(len(single_corpus)):
            feature = single_corpus[j]
            feature_id = word_id[feature]
            X[i, feature_id] = vob[feature]
    X = X.astype(int)
    matrix = X / len(vob)
    transpose = np.transpose(matrix)
    return transpose
```

首先，计算词语出现次数，保存在`vob`中，并赋予词语id，保存在`word_id`中。

然后生成矩阵，最终返回的矩阵，第i行为word_id=i的词的词向量。

TF-IDF

IDF（Inverse Document Frequency）：逆文档频率，首先我们回想一下停词，它们往往会在文档中非常高频的出现但是反而不能表达出文档的真实意思。那么同样的在不是停词的另外一些单词中，有些单词往往可以更加体现出文章的真实表达的意思，就像this thing made in china，and this thing is big。中thing只是个指代它既不能告诉你它是什么具体的东西也不能告诉你它的任何具体特征，但是big和china却可以很好的描述这句话说了什么，但是things的词频要比china和big都要大，这显然是有问题的。所以为了能够解决这么一个问题，我们需要对前面的TF进行修正，于是提出了逆文档频率，它的大小和一个词的常见程度是成反比的。

IDF的计算公式如下：

IDF = log (n / （docs (w, D)+1））

即 某一特定词语的IDF=log（总文件数目/（包含该词语的文件的数目+1））

TF-IDF：

TF-IDF=TF * IDF

某一特定文件内的高词语频率，以及该词语在整个文件集合中的低文件频率，可以产生出高权重的TF-IDF。因此，TF-IDF倾向于过滤掉常见的词语，保留重要的词语。

创建TF-IDF类来完成计算TF-IDF的相关函数：

```
class TFIDF(object):

    def __init__(self, corpus):
        """
        初始化
        self.vob:词汇个数统计，dict格式
        self.word_id:词汇编码id，dict格式
        :param corpus:输入的语料
        """
        self.word_id = {}
        self.vob = {}
        self.corpus = corpus

    def get_vob_fre(self):
        """
        计算文本频率term frequency，词id
        :return: 修改self.vob也就是修改词频统计字典
        """
        # 统计各词出现个数
        id = 0
        for single_corpus in self.corpus:
            if isinstance(single_corpus, list):
                pass
            if isinstance(single_corpus, str):
                single_corpus = single_corpus.strip("\n").split(" ")
            for word in single_corpus:
                if word not in self.vob:
                    self.vob[word] = 1
                    self.word_id[word] = id
                    id += 1
                else:
                    self.vob[word] += 1

        # 生成矩阵
        X = np.zeros((len(self.corpus), len(self.vob)))
        for i in range(len(self.corpus)):
            if isinstance(self.corpus[i], str):
                single_corpus = self.corpus[i].strip("\n").split(" ")
            else:
                single_corpus = self.corpus[i]
            for j in range(len(single_corpus)):
                feature = single_corpus[j]
                feature_id = self.word_id[feature]
                X[i, feature_id] = self.vob[feature]
        return X.astype(int)  # 需要转化成int
```

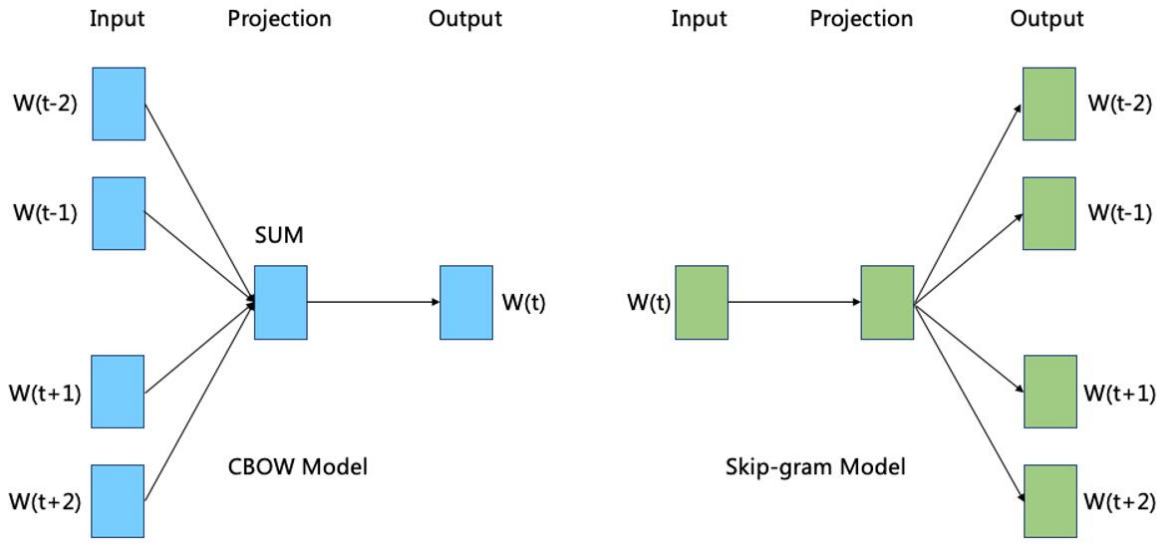
```
    def get_tf_idf(self):
        """
        计算idf并生成最后的TFIDF矩阵
        :return:
        """
        X = self.get_vob_fre()
        n_samples, n_features = X.shape
        df = []
        for i in range(n_features):
            # 这里是统计每个特征的非0的数量，也就是逆文档频率指数的分式中的分母，是为了计算
idf
            df.append(n_samples - np.bincount(X[:, i])[0])
        df = np.array(df)
        df += int(self.smooth_idf)
        n_samples += int(self.smooth_idf)
        idf = np.log(n_samples / df) + 1
        matrix = X*idf/len(self.vob)
        transpose = np.transpose(matrix)
        return transpose
```

`get_vob_fre`中，计算词语出现次数，保存在`vob`中，并赋予词语id，保存在`word_id`中，返回词出现次数的矩阵X。

`get_tf_idf`计算TFID矩阵，最后返回一个行数为词语总数，列数为50000（review总数）的矩阵，第i个行向量对应word_id为i的词语的词向量。

word2vec

Word2Vec有两个重要的模型CBOW(Continues bags of words)和Skip-gram模型。这两个模型的示意图如下：



两种模型的区别如下：

Model	如何预测	使用语料的规模
CBOW	使用上下文去预测目标词来训练得到词向量	小型语料库比较适合
Skip-gram	使用目标词去预测周围词来训练得到词向量	在大型的语料上面表现得比较好

由于我们这里的语料库较大，所以采用skip-gram模型。

生成50维的词向量。

代码如下：

```
def get_word2vec(cor):
    print('start')
    g = open("data.txt", mode='w', encoding='utf-8')
    for rev in cor:
        g.write(rev)
    g.close()

    sentences = word2vec.LineSentence('data.txt')
    model = word2vec.Word2Vec(sentences, sg=1, min_count=1, vector_size=50,
window=5)
    print('over')
    model.wv.save_word2vec_format('Embedding.txt', binary=False)
    model.save('word2vec.model')
    return model
```

将所有review（用空格将词语之间分开）写入`data.txt`中

`word2vec.LineSentence`读取`data.txt`，存储到`sentences`中，作为要训练的语料

`word2vec.Word2Vec`训练模型，其中用到的参数信息如下：

```
# 第一个参数代表要训练的语料
# sg=1 表示使用Skip-Gram模型进行训练
# vector_size 表示特征向量的维度，默认为100。大的size需要更多的训练数据,但是效果会更好. 推荐
值为几十到几百。
# window 表示当前词与预测词在一个句子中的最大距离是多少
# min_count 可以对字典做截断. 词频少于min_count次数的单词会被丢弃掉, 默认值为5
```

将训练好的model保存到`Embedding.txt`中，便于查看

并将训练好的model保存到`word2vec.model`，便于生成文件后，可以直接加载model，避免每次运行进行重复操作

CNN

本次实验分类方法选择卷积神经网络（CNN）

选择TextCNN模型来完成。

TextCNN是利用卷积神经网络对文本进行分类的算法，由Yoon Kim在其2014年的论文Convolutional Neural Networks for Sentence Classification中提出。在该论文中，作者开创性地将源于计算机视觉领域的卷积神经网络CNN应用于NLP的文本分类任务中，提出了TextCNN模型，该模型在与多个benchmark方法的对比中取得了最好的结果，成为文本分类任务的重要baseline之一。

TextCNN模型的结构比较简单，由`输入表征 --> 卷积层 --> 最大池化 --> 全连接层 --> 输出softmax`组成, 如图所示：

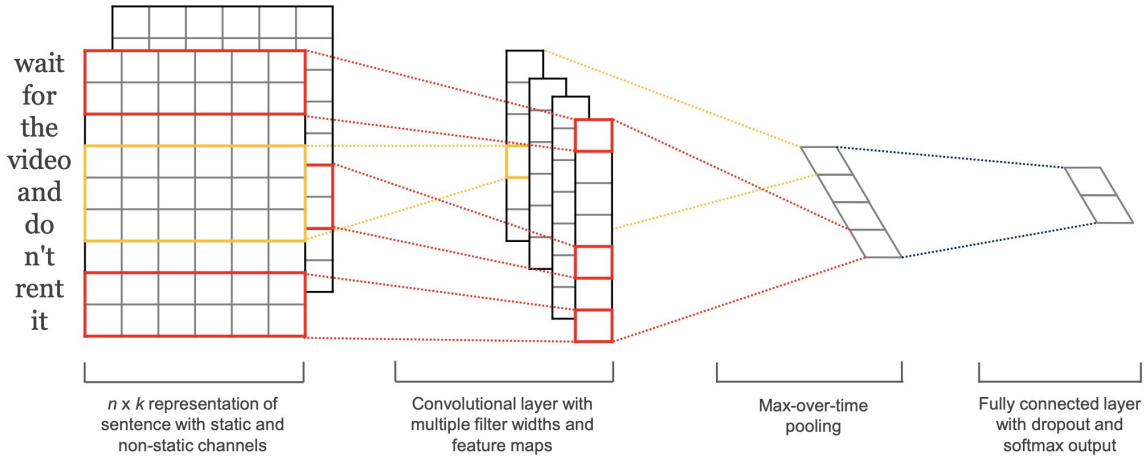


Figure 1: Model architecture with two channels for an example sentence.

输入表征：

由于CNN的输入要求是固定尺寸的（全连接层的限制），因此输入文本需要转换为n × k 的矩阵形式。而且，由于计算机无法直接处理文字，因此需要对文本进行数值化，将文本信息映射到数值化的语义空间中，即常说的词向量空间。

卷积层：

在计算机视觉领域，CNN的输入是二维的图像数据，因此在卷积层进行的是二维卷积运算。而文本数据是只有一个维度的时间序列，因此只能将文本当作一维图像，在卷积层使用一维卷积运算来捕捉临近词之间的关联关系。TextCNN中的卷积层采用一维卷积操作（虽然文本经过词向量表达后是二维数据，但是在词向量维度上的二维卷积是没有意义的），一维卷积带来的问题是需要通过设计不同kernel_size的卷积滤波器来获取不同宽度的感受野。

最大池化：

TextCNN中使用的池化方法是时序最大池化（Max-over-time pooling）。它实际上就是一维的全局最大池化，即对于多通道输入，各通道的输出是所有时间步中最大的数值。使用时序最大池化还有个优势是，它可以提取时序数据中最重要的特征，使得模型不受人为添加字符的影响（在构建词向量时对于较短的句子通常使用0向量补全，使用最大池化可以过滤掉这些补0的数据）。

全连接层：

数据经过池化处理后，作为输入送入全连接层，全连接层起到将学到的特征表示映射到样本标记空间的作用。在实际应用中为了防止过拟合，通常在全连接层加dropout来提升模型的泛化能力。

输出softmax：

全连接层的输出是0到1之间的概率值，要解决分类问题需要使用softmax函数来将这些概率转化为离散的0或1类标。

输入数据处理：将句子转换为编号列表

我们需要将词语转化为方便计算机处理的数字，最直观的想法就是给每个词分一个编号，以编号来代替词语。

词典vocab的构建已经在文本的预处理中完成。

我们为vocab中的词语编号：

```
word2idx = {i: index for index, i in enumerate(vocab)}
idx2word = {index: i for index, i in enumerate(vocab)}
```

有了`word2idx`和`idx2word`，就可以轻松的在词语和编号之间转换。

由此，我们可以将一个句子用一个编号列表来表示，这样就可以很方便的交给计算机处理。

但是这里还存在2个问题：

1. 我们的vocab不存在低频词，我们没有对低频词进行编号，那这些词要怎么转为数字呢？
2. 原句子长短不一，转换后的编号列表也长短不齐，但是后续模型的输入需要固定尺寸

这两个问题都可以通过对编号列表进行补全/截断操作来解决。对于低频词，我们选取一个与情感无关的高频中性词`two`来代替它的位置，这样也不会影响到对句子情感的判断。我们选取一个固定句子长度`sequence_length`，对于小于其的句子，用`two`来填充缺少的长度，对于大于其的句子，简单的进行截断即可。

对于`sequence_length`的选取，需要参考语料库中review的长度情况。这里计算每条review的长度，并输出相关数据：

```
# 计算review长度，好选定截断review的长度，这里选择286
comments_len = train_data.iloc[:, 1].apply(lambda x: len(x.split()))
train_data["comments_len"] = comments_len
print(train_data["comments_len"].describe(percentiles=[.5, .95]))
```

输出：

```
count    50000.000000
mean       112.042240
std         83.163856
min          2.000000
50%         84.000000
95%        286.000000
max       1172.000000
Name: comments_len, dtype: float64
```

可以看到，95%的review的长度≥286，为尽可能保留句子内容，故这里选择令`sequence_length = 286`

设计`tokenizer`函数来完成句子到编号列表的转换，包括进行截断/补全：

```
# 对句子用编号表示，对句子进行截断/补全
pad_id = word2idx["two"]# 填充使用”two“(无情感词)
sequence_length = 286# 句子固定长度


def tokenizer():
    inputs = []
    sentence_char = [i.split() for i in train_data["review"]]
```

```
    for index, i in enumerate(sentence_char):
        # 转换为编号列表
        temp = [word2idx.get(j, pad_id) for j in i]
        if len(i) < sequence_length:
            # 补全
            for _ in range(sequence_length-len(i)):
                temp.append(pad_id)
        else:
            # 截断
            temp = temp[:sequence_length]
        inputs.append(temp)
    return inputs


data_input = tokenizer()
```

参数定义

```
# 参数定义
device = "cpu"
Embedding_size = 250
Batch_Size = 64
Kernel = 3
Filter_num = 10# 卷积核的数量。
Epoch = 30
Dropout = 0.5
Learning_rate = 1e-3
num_classs = 2# 2分类问题
```

准备数据集

```
class TextCNNDataSet(Data.Dataset):
    def __init__(self, data_inputs, data_targets):
        self.inputs = torch.LongTensor(data_inputs)
        self.label = torch.LongTensor(data_targets)

    def __getitem__(self, index):
        return self.inputs[index], self.label[index]

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


# 划分数据集
TextCNNDataSet = TextCNNDataSet(data_input, list(train_data["sentiment"]))
train_dataset = torch.utils.data.Subset(TextCNNDataSet, range(0, 30000))
val_dataset = torch.utils.data.Subset(TextCNNDataSet, range(30000, 40000))
test_dataset = torch.utils.data.Subset(TextCNNDataSet, range(40000, 50000))

TrainDataLoader = Data.DataLoader(train_dataset, batch_size=Batch_Size,
shuffle=True)
```

```
TestDataLoader = Data.DataLoader(test_dataset, batch_size=Batch_Size,
shuffle=True)
ValDataLoader = Data.DataLoader(val_dataset, batch_size=Batch_Size,
shuffle=True)
```

定义`TextCNNDataSet`类，管理数据集的设置。

利用`torch.utils.data.Subset`可以按照提供的下标划分数据集。

用Data.DataLoader（torch.utils.data）数据加载器，组合数据集和采样器，并在数据集上提供单进程或多进程迭代器。它可以对数据集作进一步的设置(比如可以设置打乱，对数据裁剪，设置batch_size等操作)

这里我们按照要求，划分0-30000条为训练集、30001-40000条为验证集和40001-50000条为测试集，并在各个数据集内部打乱顺序。

加载词向量

tf

原tf词向量维度为50000，过大，输入到网络中需要很多内存，无法执行。所以这里使用PCA为其降维。

用于降维的函数：

```
def mypca(w, dimension):
    pca = PCA(n_components=dimension)  # 初始化PCA
    X = pca.fit_transform(w)  # 返回降维后的数据
    return X
```

这里降维到

```
corpus = train_data['review']
# tf作为向量
w2v = get_tf(corpus)
w2v = mypca(w2v,250)
```

由于我们最终输入的是word的编号，所以这里我们需要将w2v转换成每个word编号来对应其vector：

```
def word2vec(x):
    # x是编号的形式
    x2v = np.ones((len(x), x.shape[1], Embedding_size))
    for i in range(len(x)):
        temp = []
        for j in x[i]:
            w = idx2word[j.item()]
            k = word_id[w]
            temp.append(w2v[k])
        x2v[i] = temp
    return torch.tensor(x2v).to(torch.float32)
```

x为一个batch，即包含多个句子。

x2v为三维矩阵，它存储batch中每个句子中的每个词对应向量。

对于句子中每个词编号j，我们通过`idx2word[j.item()]`获取词语本身，再通过tf的word_id，获得该词的词向量在tf矩阵中所在的行数，则该行即为该词向量。

tf-idf

tf-idf和tf的形式相似，所以操作流程也相似。使用相同的`mypca`函数降维到，再用相同的`word2vec`函数，将word编号和其vector对应起来。

获得（加载）tf-idf词向量：

```
corpus = train_data['review']
# tfidf作为向量
tfidf = TFIDF(corpus)
w2v = tfidf.get_tf_idf()
w2v = mypca(w2v, 250)
```

word2vec

```
# 载入word2vec作为向量
get_word2vec(corpus)
w2v = word2vec.Word2Vec.load('word2vec.model')
```

w2v存储的是每个word对应的vector，由于我们最终输入的是word的编号，所以这里我们需要将w2v转换成每个word编号来对应其vector：

```
def word2vec(x):
    # x是编号的形式
    x2v = np.ones((len(x), x.shape[1], Embedding_size))
    for i in range(len(x)):
        x2v[i] = w2v.wv[[idx2word[j.item()] for j in x[i]]]
    return torch.tensor(x2v).to(torch.float32)
```

定义网络

```
class TextCNN(nn.Module):
    def __init__(self):
        super(TextCNN, self).__init__()
        out_channel = Filter_num # 可以等价为通道的解释。
        self.conv = nn.Sequential(
                    nn.Conv2d(1, out_channel, (2, Embedding_size)),# 卷积核大小为2*Embedding_size
                    nn.ReLU(),
                    nn.MaxPool2d((sequence_length-1, 1)),
        )
        self.dropout = nn.Dropout(Dropout)
        self.fc = nn.Linear(out_channel, num_classs)

    def forward(self, X):
        batch_size = X.shape[0]
        embedding_X =  word2vec(X)
```

```
        embedding_X = embedding_X.unsqueeze(1)
        conved = self.conv(embedding_X)
        conved = self.dropout(conved)
        flatten = conved.view(batch_size, -1)
        output = self.fc(flatten)
        # 2分类问题，往往使用softmax，表示概率。
        return F.log_softmax(output)
```

卷积层：

卷积的作用就是提取特征，因为一次卷积可能提取的特征比较粗糙，所以多次卷积，以及层层纵深卷积，层层提取特征（千万要区别于多次卷积，因为每一层里含有多次卷积）。

我们的词向量为50维，对于一个长度为500的句子，便可以得到500行50列的矩阵A。我们可以把矩阵A看成是一幅图像，使用卷积神经网络去提取特征。这里我们用`nn.Conv2d`来实现二维卷积。

此函数的形参由Pytorch手册可以查得，前三个参数是必须手动提供的，后面的有默认值：

```
CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0,
      dilation=1, groups=1, bias=True, padding_mode='zeros')
```

可以看到，在Pytorch的nn模块中，不需要手动定义网络层的权重和偏置。在nn模块中，Pytorch对于卷积层的权重和偏置（如果需要偏置）初始化都是采用He初始化的，因为它非常适合于ReLU函数。

函数中的参数说明如下：

in_channels	Number of channels in the input image
out_channels	Number of channels produced by the convolution
kernel_size	卷积核尺寸
stride	步长，控制cross-correlation的步长，可以设为1个int型数或者一个(int, int)型的tuple。
padding	(补0)：控制zero-padding的数目。
dilation	(扩张)：控制kernel点（卷积核点）的间距
groups	(卷积核个数)：通常来说，卷积个数唯一，但是对某些情况，可以设置范围在1 —— in_channels中数目的卷积核：
bias	adds a learnable bias to the output.

这里，我们设置`in_channels=1,out_channel=Filter_num=10,kernel_size=2*Embedding_size`，其他都使用默认值。

激活函数

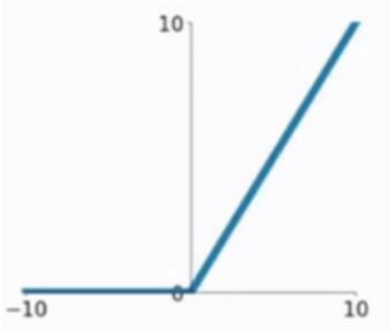
激活函数的作用是将线性分布转化为非线性分布。

如果不用激活函数，每个网络层的输出都是一种线性输出，而易知，我们本次的问题显然是非线性的分布。

这里使用ReLU函数作为激活函数。ReLU函数不存在梯度消失问题，且在 x > 0 时，Relu求导 = 1，这对于反向传播计算dw，db，是能够大大的简化运算的。

ReLU函数公式和曲线如下

$$f(x) = max(0, x)$$



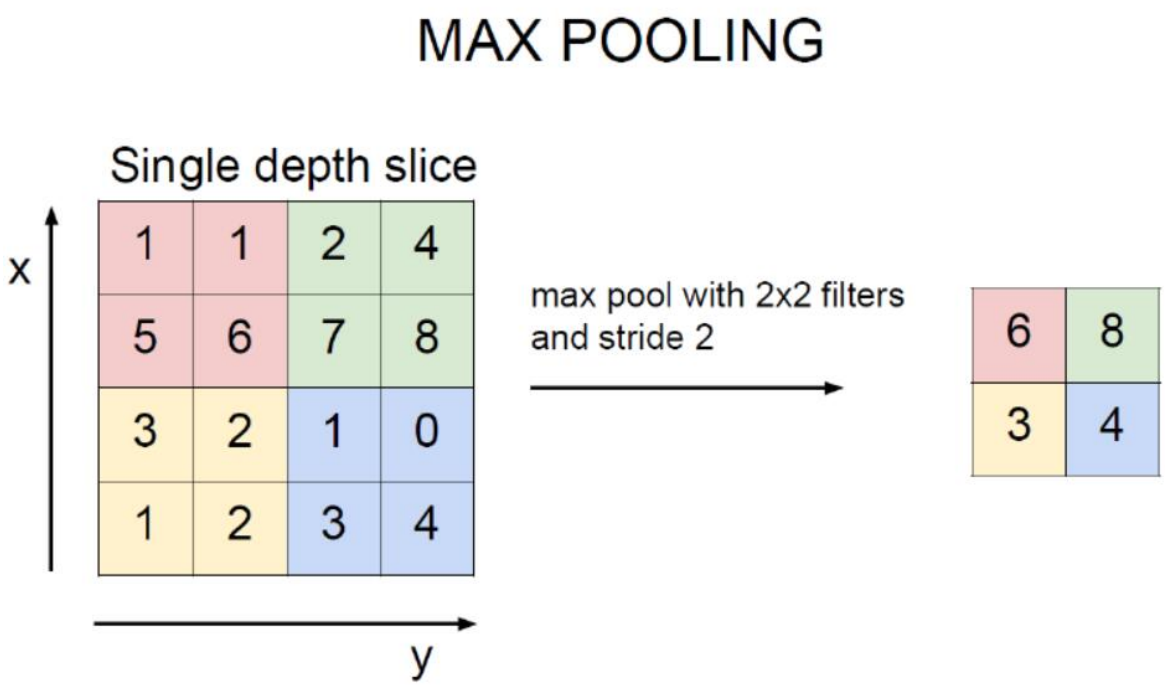
池化

池化层一般在卷积层+ Relu之后，它的作用是：

1、减小输入矩阵的大小（只是宽和高，而不是深度），提取主要特征。（不可否认的是，在池化后，特征会有一定的损失，所以，有些经典模型就去掉了池化这一层）。它的目的是显而易见的，就是在后续操作时能降低运算。

2、一般采用mean_pooling（均值池化）和max_pooling（最大值池化），对于输入矩阵有translation（平移），rotation（旋转），能够保证特征的不变性。

这里我们采用max_pooling 最大值池化，即每个池化区域的最大值放在输出对应位置上。



全连接层

全连接层是作为分类器角色，将特征映射到样本标记空间，本质是矩阵变换（affine）。

使用`nn.linear()`来设置网络中的全连接层的。在全连接层中的输入与输出都是二维张量，一般形状为[batch_size, size]，与卷积层要求输入输出是4维张量不同。

使用`nn.Dropout(Dropout)`,防止出现过拟合。

前向传播

这里定义`forward`函数完成前向传播。

前向传播包含之前的卷积，Relu激活函数，池化（pool），全连接(fc)，可以说，在损失函数之前操作都属于前向传播。主要是权重参数w , b 初始化，迭代，以及更新w, b,生成分类器模型。

使用`word2vec`作为`embedding`。

我们使用`softmax`进行分类。将 max-pooling的结果拼接起来, 送入到softmax当中, 得到label 为1的概率以及label 为0的概率。如果是预测的话，到这里整个textCNN的流程遍结束了。如果是训练的话，此时便会根据预测label以及实际label来计算损失函数, 计算出softmax 函数,max-pooling 函数, 激活函数以及卷积核函数四个函数当中参数需要更新的梯度, 来依次更新这四个函数中的参数，完成一轮训练。

训练网络

```
model = TextCNN().to(device)
optimizer = optim.Adam(model.parameters(), lr=Learning_rate)

def binary_acc(pred, y):
    """
    计算模型的准确率
    :param pred: 预测值
    :param y: 实际真实值
    :return: 返回准确率
    """
    correct = torch.eq(pred, y).float()
    acc = correct.sum() / len(correct)
    return acc.item()

def train():
    avg_acc = []
    model.train()
    for index, (batch_x, batch_y) in enumerate(TrainDataLoader):
        batch_x, batch_y = batch_x.to(device), batch_y.to(device)
        pred = model(batch_x)
        loss = F.nll_loss(pred, batch_y)
        acc = binary_acc(torch.max(pred, dim=1)[1], batch_y)
        avg_acc.append(acc)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    avg_acc = np.array(avg_acc).mean()
    return avg_acc

def validation():
    """
    模型验证
    :param model: 使用的模型
    :return: 返回当前训练的模型在验证集上的结果
    """
    avg_acc = []
    model.eval()  # 进入测试模式
    with torch.no_grad():
        for x_batch, y_batch in ValDataLoader:
            x_batch, y_batch = x_batch.to(device), y_batch.to(device)
```

```
            pred = model(x_batch)
            acc = binary_acc(torch.max(pred, dim=1)[1], y_batch)
            avg_acc.append(acc)
    return np.array(avg_acc).mean()


# 训练
model_train_acc, model_test_acc = [], []
print("开始训练...")
for epoch in range(Epoch):
    train_acc = train()
    print("epoch = {}, 训练准确率={}".format(epoch + 1, train_acc))
    if (epoch+1) % 10 == 0:
        val_acc = validation()
        print("epoch = {}, 验证集准确率={}".format(epoch + 1, val_acc))
    model_train_acc.append(train_acc)

torch.save(model, 'save.pt')
checkpoint = {"model_state_dict": model.state_dict(),
              "optimizer_state_dict": optimizer.state_dict(),
              "epoch": Epoch}
path_checkpoint = "./checkpoint_{}_epoch.pkl".format(Epoch)
torch.save(checkpoint, path_checkpoint)
```

总共进行30轮epoch，每10轮，进行一次验证。

将最终训练出来的模型保存到`save.pt`,并保存`checkpoint`。

测试

```
def evaluate():
    """
    模型评估
    :param model: 使用的模型
    :return: 返回当前训练的模型在测试集上的结果
    """
    avg_acc = []
    avg_pre = []
    avg_recall = []
    avg_f1 = []
    model.eval()  # 进入测试模式
    with torch.no_grad():
        for x_batch, y_batch in TestDataLoader:
            x_batch, y_batch = x_batch.to(device), y_batch.to(device)
            pred = model(x_batch)
            y_pred = torch.max(pred, dim=1)[1]
            acc = binary_acc(y_pred, y_batch)
            pre = sklearn.metrics.precision_score(y_batch, y_pred)
            recall = sklearn.metrics.recall_score(y_batch, y_pred)
            f1 = sklearn.metrics.f1_score(y_batch, y_pred)
            avg_acc.append(acc)
            avg_pre.append(pre)
            avg_recall.append(recall)
            avg_f1.append(f1)
    eval_acc = np.array(avg_acc).mean()
    eval_pre = np.array(avg_pre).mean()
```

```
    eval_recall = np.array(avg_recall).mean()
    eval_f1 = np.array(avg_f1).mean()
    print("测试集准确率={}".format(eval_acc))
    print("测试集精确率={}".format(eval_pre))
    print("测试集召回率={}".format(eval_recall))
    print("测试集F1-Score={}".format(eval_f1))


evaluate()
```

这里对训练结果的评价使用了准确率、精确率、召回率、F1值，4种评价标准。

通常以关注的类为正类，其他类为负类，分类器在数据集上的预测或者正确或者不正确，我们有4种情况，在混淆矩阵中表示如下：

精确率： P = TP / (TP + FP)，其含义是在被所有预测为正的样本中实际为正样本的概率

召回率： R = TP / (TP + FN)，其含义是在被所有实际为正的样本中实际为正样本的概率

F1: 精确率和召回率的调和平均。 即： 2/F1 = 1/P + 1/R

可视化

```
# 展示训练过程中准确率变化
plt.plot(model_train_acc)
plt.ylim(ymin=0.5, ymax=1)
plt.title("The accuracy of textCNN model")
plt.show()
```

结果分析

tf

运行程序，输出：

```
epoch = 1, 训练准确率=0.6139392324093816
epoch = 2, 训练准确率=0.6655561477899044
epoch = 3, 训练准确率=0.6866449004551495
epoch = 4, 训练准确率=0.6960065742291367
epoch = 5, 训练准确率=0.7002154406962364
epoch = 6, 训练准确率=0.706856343283582
epoch = 7, 训练准确率=0.7020255863539445
epoch = 8, 训练准确率=0.7121090974126544
epoch = 9, 训练准确率=0.7119869402985075
epoch = 10, 训练准确率=0.7146632906470471
epoch = 10, 验证集准确率=0.7590565286624203
epoch = 11, 训练准确率=0.7162735430416522
epoch = 12, 训练准确率=0.7191164712153518
epoch = 13, 训练准确率=0.7192608386214608
epoch = 14, 训练准确率=0.7233031272634006
epoch = 15, 训练准确率=0.7273898364892647
epoch = 16, 训练准确率=0.7303771321961621
epoch = 17, 训练准确率=0.728578091684435
epoch = 18, 训练准确率=0.7327203268943819
epoch = 19, 训练准确率=0.72882240578564
epoch = 20, 训练准确率=0.7346748400852878
epoch = 20, 验证集准确率=0.7783638535031847
```

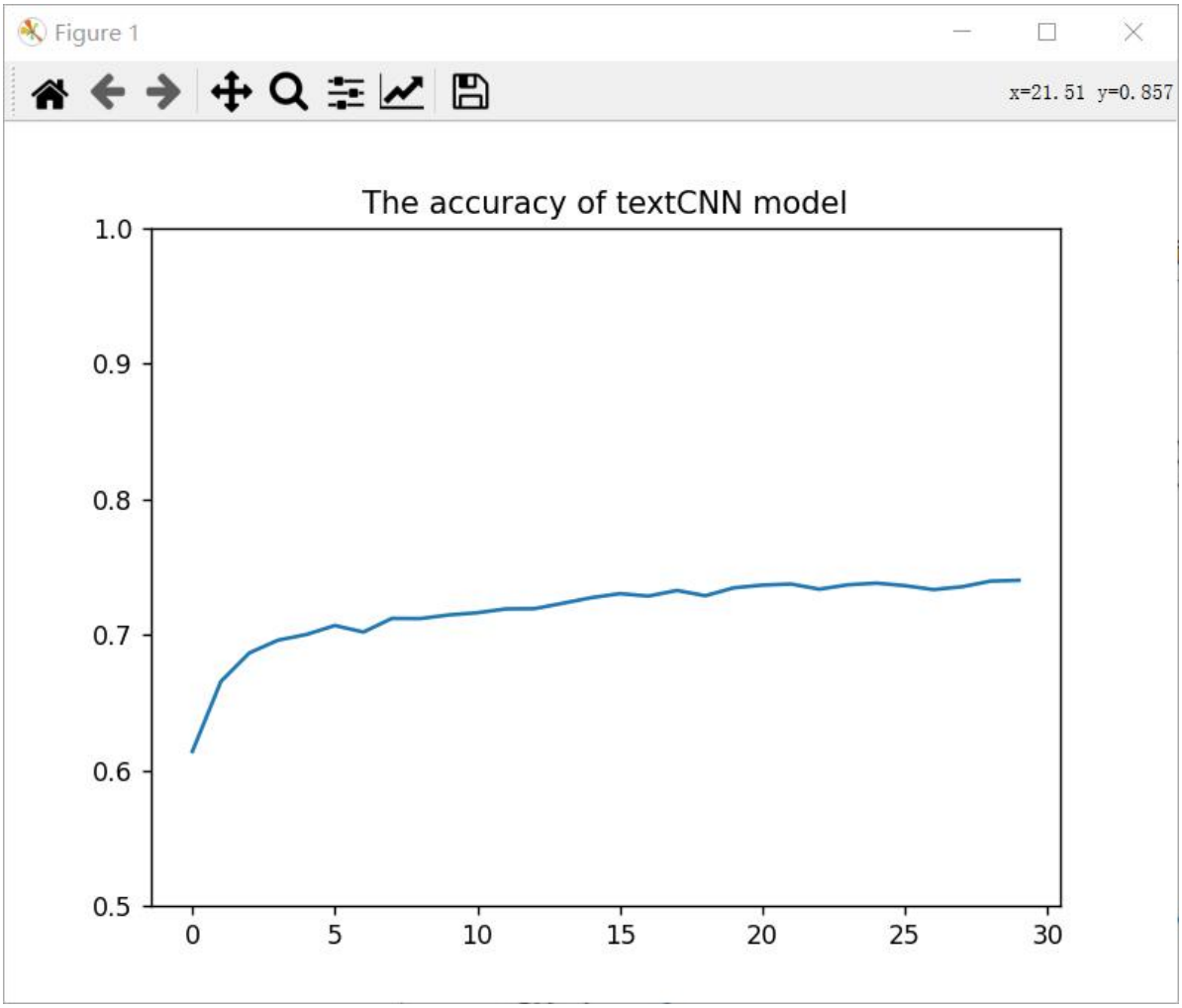
```
epoch = 21, 训练准确率=0.7366515635681559
epoch = 22, 训练准确率=0.7374622424019933
epoch = 23, 训练准确率=0.7336531627152775
epoch = 24, 训练准确率=0.7368847725233798
epoch = 25, 训练准确率=0.7381396588486141
epoch = 26, 训练准确率=0.7362517767877721
epoch = 27, 训练准确率=0.7333644279030594
epoch = 28, 训练准确率=0.7354188876619725
epoch = 29, 训练准确率=0.7395833332909704
epoch = 30, 训练准确率=0.7402052238805971
epoch = 30, 验证集准确率=0.785031847133758
```

可以看到，每进行10次epoch，进行一次验证，3次验证集上的准确率分别为75.91%、77.84%、78.50%

```
测试集准确率=0.7855294585987261
测试集精确率=0.7685197959297991
测试集召回率=0.8193246352125438
测试集F1-Score=0.7901911847461516
```

最终，测试集上的准确率为78.55%，精确率为76.85%，召回率为81.93%，F1-Score为79.02%

下图为训练集上准确率的变化曲线：



tf-idf

运行程序，输出：

```
epoch = 1, 训练准确率=0.595726723482868
epoch = 2, 训练准确率=0.6559501599147122
epoch = 3, 训练准确率=0.6828913468796053
epoch = 4, 训练准确率=0.697117093148262
epoch = 5, 训练准确率=0.7047574626865671
epoch = 6, 训练准确率=0.712897565700352
epoch = 7, 训练准确率=0.713064143525512
epoch = 8, 训练准确率=0.7179393212932513
epoch = 9, 训练准确率=0.7244136460554371
epoch = 10, 训练准确率=0.7226257106896911
epoch = 10, 验证集准确率=0.7660230891719745
epoch = 11, 训练准确率=0.7214929814786036
epoch = 12, 训练准确率=0.7212708777964496
epoch = 13, 训练准确率=0.7198494136460555
epoch = 14, 训练准确率=0.7288668266237418
epoch = 15, 训练准确率=0.7294442963752665
epoch = 16, 训练准确率=0.7249466950959488
epoch = 17, 训练准确率=0.7299440298507462
epoch = 18, 训练准确率=0.7302327647900531
epoch = 19, 训练准确率=0.732620380199286
epoch = 20, 训练准确率=0.7319651741717161
epoch = 20, 验证集准确率=0.7734872611464968
```

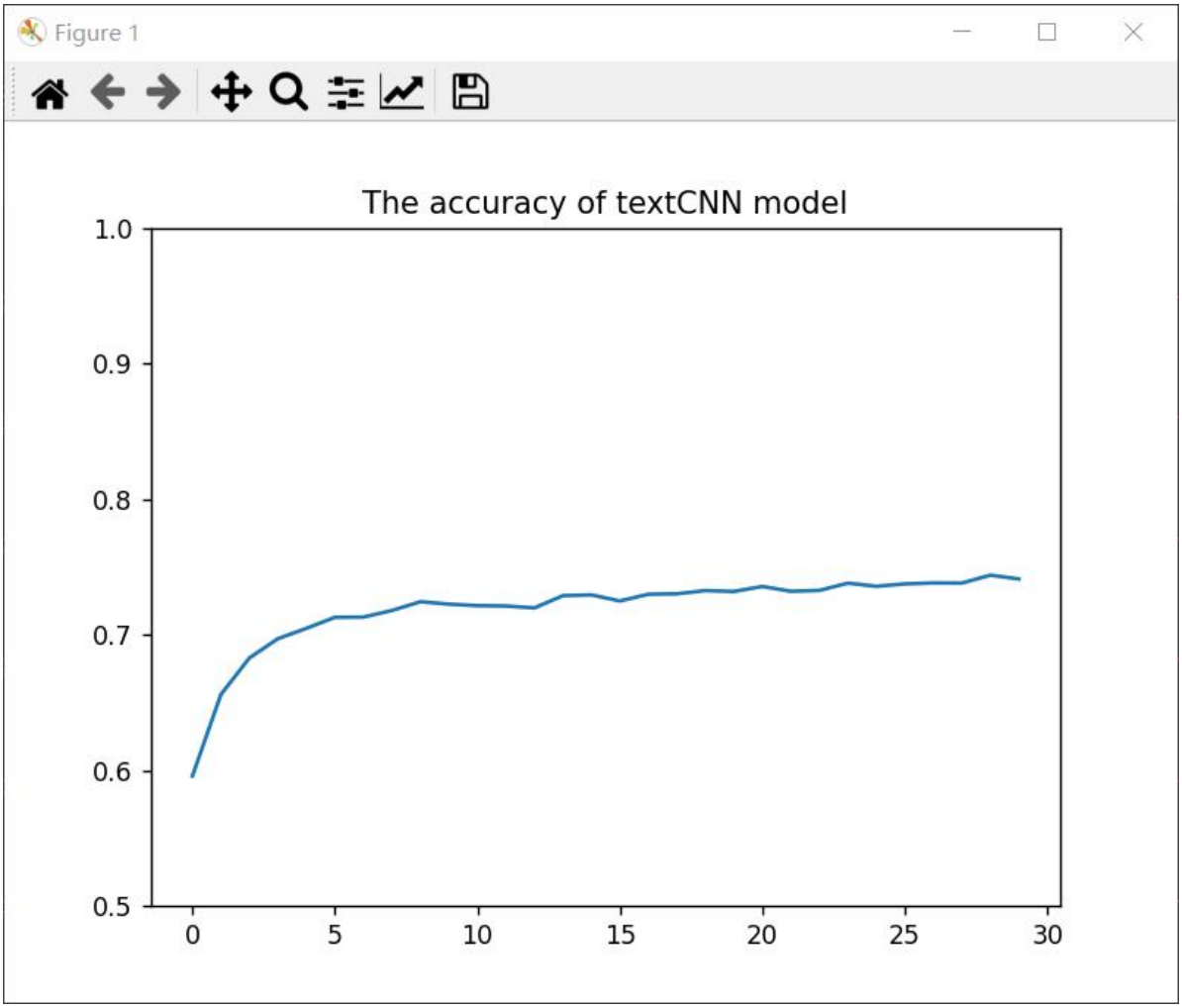
```
epoch = 21, 训练准确率=0.7356520966171964
epoch = 22, 训练准确率=0.7320984364318441
epoch = 23, 训练准确率=0.7327425373134329
epoch = 24, 训练准确率=0.7381174484295632
epoch = 25, 训练准确率=0.7357742537313433
epoch = 26, 训练准确率=0.7375621890970893
epoch = 27, 训练准确率=0.738272921108742
epoch = 28, 训练准确率=0.738195184832697
epoch = 29, 训练准确率=0.743992093148262
epoch = 30, 训练准确率=0.7412046908315565
epoch = 30, 验证集准确率=0.7803542993630573
```

可以看到，每进行10次epoch，进行一次验证，3次验证集上的准确率分别为76.60%、77.35%、78.04%

```
测试集准确率=0.7810509554140127
测试集精确率=0.7471911600862831
测试集召回率=0.8510673946052256
测试集F1-Score=0.7936007142175533
```

最终，测试集上的准确率为78.11%，精确率为74.72%，召回率为85.11%，F1-Score为79.36%

下图为训练集上准确率的变化曲线：



word2vec

运行程序，输出：

```
epoch = 1, 训练准确率=0.6680881307323335
epoch = 2, 训练准确率=0.7763415067943175
epoch = 3, 训练准确率=0.7866804371002132
epoch = 4, 训练准确率=0.7920886637813755
epoch = 5, 训练准确率=0.7937100213219617
epoch = 6, 训练准确率=0.7922219260415034
epoch = 7, 训练准确率=0.7963641613785392
epoch = 8, 训练准确率=0.7936544953378787
epoch = 9, 训练准确率=0.8001843460841473
epoch = 10, 训练准确率=0.8020833332909704
epoch = 10, 验证集准确率=0.8398686305732485
epoch = 11, 训练准确率=0.8055259417623345
epoch = 12, 训练准确率=0.8022054904051172
epoch = 13, 训练准确率=0.8042155294530173
epoch = 14, 训练准确率=0.8029273276898399
epoch = 15, 训练准确率=0.8054704157782516
epoch = 16, 训练准确率=0.8015280739584966
epoch = 17, 训练准确率=0.8042821605830813
epoch = 18, 训练准确率=0.804626421506471
epoch = 19, 训练准确率=0.8035047974413646
epoch = 20, 训练准确率=0.8008284470928249
epoch = 20, 验证集准确率=0.8454418789808917
```

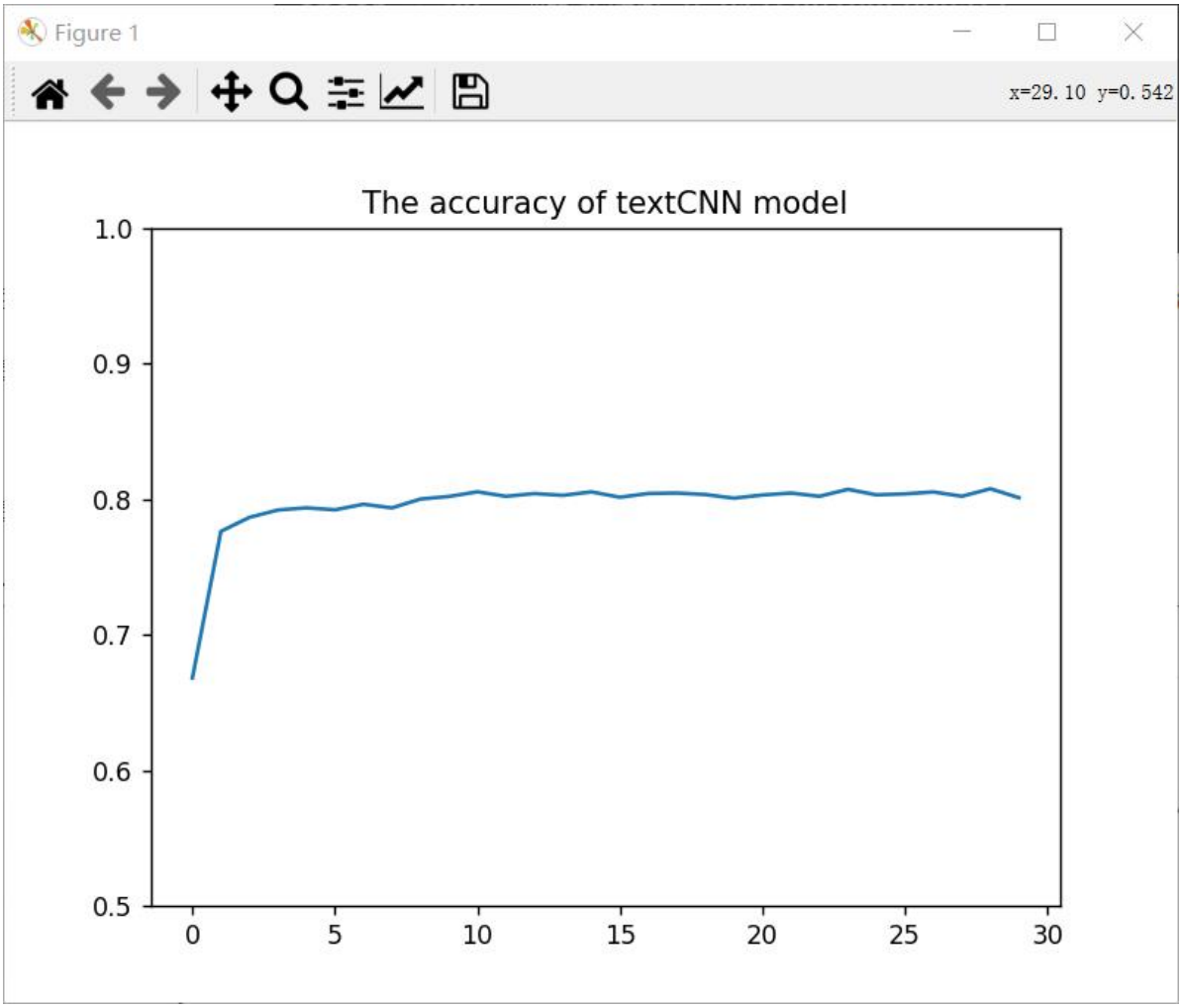
```
epoch = 21, 训练准确率=0.8031605366450637
epoch = 22, 训练准确率=0.8046153162334011
epoch = 23, 训练准确率=0.8022054904051172
epoch = 24, 训练准确率=0.8073471925660237
epoch = 25, 训练准确率=0.8032715884861408
epoch = 26, 训练准确率=0.8039934257708633
epoch = 27, 训练准确率=0.8054148897941686
epoch = 28, 训练准确率=0.8022499111161303
epoch = 29, 训练准确率=0.8077802949114394
epoch = 30, 训练准确率=0.8011727078891258
epoch = 30, 验证集准确率=0.8451433121019108
```

可以看到，每进行10次epoch，进行一次验证，3次验证集上的准确率分别为83.99%、84.54%、84.51%

```
测试集准确率=0.8539012738853503
测试集精确率=0.8509083945637427
测试集召回率=0.8587918815609891
测试集F1-Score=0.853160748934028
```

最终，测试集上的准确率为85.39%，精确率为85.09%，召回率为85.88%，F1-Score为85.32%

下图为训练集上准确率的变化曲线：



3者之间的比较

tf、tf-idf、word2vec三种特征，使用tf、tf-idf的250维词向量，word2vec的50维词向量，由同样的网络进行学习训练，均训练30个epoch。

下面我们来简单比对下它们3个的训练结果：

收敛速度：

通过训练集上准确率的变化曲线，可以明显看出，word2vec的收敛速度是最快的。tf和tf-idf收敛速度差不多。

准确率：

训练集、验证集、测试集上的准确率，均为word2vec>tf-idf≈tf

精确率：

word2vec>tf>tf-idf

召回率：

word2vec>tf-idf>tf

F1-Score：

word2vec>tf-idf≈tf

tf和tf-idf的表现差不多，这个是有点超出了我的预料，因为tf-idf是tf的改进版。经过思考后，考虑到应该是因为我做了去除高频词的处理，所以一定程度上消除了tf的此类弊端：有些常用词在语料库中出现频率很高，但是它们对目标变量的预测能力却很小，而真正能预测的关键词语的出现频率可能很低，但是它仅出现一两次便可决定整个文本的情感。通过去除高低频词，使得这种干扰的常用词被去除，所以可以说由某个词在某类情感下出现的频率来判断其情感色彩是可靠的。

但是在本次实验中，tf和tf-idf的表现一般，考虑可能是由于下面的原因：

1. idf是一种试图抑制噪声的加权，本身倾向于文本中频率比较小的词，这使得idf的精度不高
2. tf-idf比较依赖于语料库（尤其在训练同类语料库时，往往会掩盖一些同类型的关键词；如：在进行 *TF-IDF* 训练时，语料库中的 娱乐 新闻较多，则与 娱乐 相关的关键词的权重就会偏低 ），因此需要选取质量高的语料库进行训练。而本次选择的语料库为IMDB评论集，其质量可能没有很高。
3. tf、tf-idf反映的词之间各自独立，无法反映序列信息

word2vec是表现最好的，大概是因为word2vec的通用性很强，在各类NLP问题中都能有较好的表现，并且相比于tf-idf，word2vec会考虑上下文信息。

本次实验中尝试了进行调参和更改预处理，但是tf和tf-idf的表现还是不是很理想。一是个人认为tf、tf-idf获得的词向量是独立的，而本次的语料库中，单个词对情感的指向性不强，大多数是词语的组合来指向情感，所以word2vec表现更好；二是TextCNN有一个缺点，就是模型可解释型不强，在调优模型的时候，很难根据训练的结果去针对性的调整具体的特征。

因此，上面所述是大体的一个情况，可能经过调优后表现情况会有些许变化。

遇到的问题及解决

nltk.download('stopwords')报错

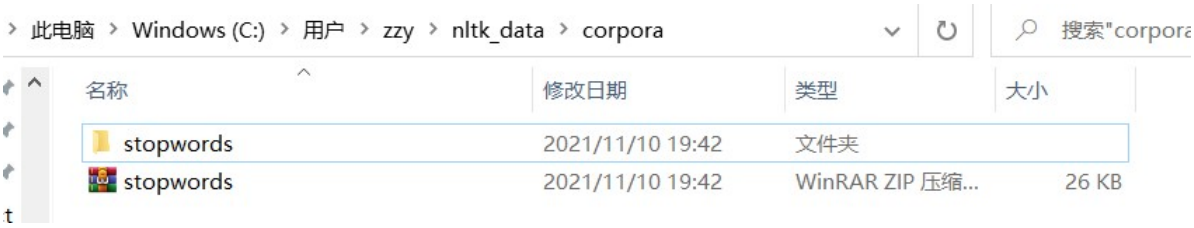
```
LookupError:
**********************************************************************
  Resource stopwords not found.
  Please use the NLTK Downloader to obtain the resource:

  >>> import nltk
  >>> nltk.download('stopwords')

  For more information see: https://www.nltk.org/data.html
```

说明无法下载stopwords，于是考虑自己手动下载stopwords

从Github上下载下stopwords.zip,并解压放在报错时，告知的search in的第一个路径下。

Github地址为 https://github.com/nltk/nltk_data/tree/gh-pages/packages/corpora
我的路径是C:\Users\zzy\nltk_data

由于报错告诉我们，在路径下search的是nltk_data\corpora，所以在该路径下创建nltk_data->corpora，将解压后的stopwords放在这下面：



再次运行，不再报错：

```
[nltk_data] Downloading package stopwords to C:\Users\zzy/nltk_data...
[nltk_data]   Unzipping corpora\stopwords.zip.
```

对数据进行重复操作

由于本次实验的数据量较大，所以若不将一些中间的结果保存下来，每次运行程序都要重新跑整个过程，这样等待出结果的时间将很长。

所以我选择将过程中，一些步骤的结果保存，第二次运行开始，直接载入保存好的结果，而不用重复进行操作。

本次实验中，保存了以下中间结果：

`clean.csv`：预处理后的数据，有review和sentiment（0表示消极，1表示积极）两列

`word.txt`：所有review中的所有词

`word_freq.txt`：去除低频词后的words

`word2vec.model`：训练后的word2vec向量

pd.read_csv多出一列

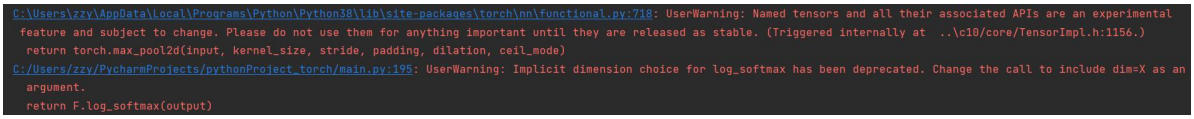
使用`dataframe.to_csv`存csv就会多出一列下标列，此时再将该csv读取后，会出现两列下标列，类似下图的情况：

	Unnamed: 0
0	0
1	1

解决方法，读取的时候，设置第0列为下标列即可：

```
# 载入预处理后的数据
train_data = pd.read_csv("clean.csv", index_col=0)
```

UserWarning



这里我遇到了2个warning：

1.

```
UserWarning: Named tensors and all their associated APIs are an experimental
feature and subject to change. Please do not use them for anything important until
they are released as stable. (Triggered internally at
..\c10/core/TensorImpl.h:1156.)
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)
```

这是pytorch1.9的bug，下个版本将修复，可以通过将pytorch降级成1.8版本来解决这个问题。

2.

```
UserWarning: Implicit dimension choice for log_softmax has been deprecated. Change
the call to include dim=X as an argument.
  return F.log_softmax(output)
```

这个警告的原因是softmax（）函数已经被弃用了，虽然程序还是可以运行成功，但是这个做法不被pytorch所赞成。这个写法在早期的pytorch版本是没有警告的，现在因为其他考虑，要加上有指明dim参数。

内存不够用

在计算tf、tf-idf矩阵的时候频频遇到内存不够用的问题：

计算矩阵的时候，显示内存不够用：

```
numpy.core._exceptions.MemoryError: Unable to allocate 38.2 GiB for an array with shape (50000, 102670) and data type float64
```

于是将截断的操作提前了，review>500的部分截掉，并增加了去除停用词的操作，让词语总数变少。

另外需注意的是，进行计算矩阵的操作时，要将别的应用（如浏览器）什么的最好都关闭，因为它们也会占用内存。

这样操作下来，勉强内存够用，下图为计算矩阵时内存占用详情：

可以看到，内存占用已经快不够用了。

训练的时候显示内存不够：

```
RuntimeError: [enforce fail at ..\c10\core\CPUAllocator.cpp:79] data. DefaultCPUAllocator: not enough memory: you tried to allocate 12800000000 bytes.
```

原因是生成的向量维度太大了，于是进行PCA降维。

心得体会

通过本次实验，我学会了如何对文本进行清洗和预处理，知道了预处理的操作大概为：去除HTML标签、去除标点符号、删除停用词、删除低频词、大小写统一等。学习了tf、tf-idf、word2vec是如何表示词向量的，并亲自实现了tf和tf-idf。同时也了解到了这三种特征表示各自的优缺点。明白了将文本输入网络的话，需要将文本转换成数字，用编号代表词语。本次实验使用的深度学习模型为CNN，熟悉了cnn的定义和内部结构，以及如何训练网络、如何使用验证集、如何进行测试。

总之，本次实验收获很大，从一开始的，不清楚tf、tf-idf如何构成词向量（不了解是各个文档下的tf/tf-idf的值构成了词向量的各个维度），到最终通过查阅资料自己写出tf、tf-idf，成就感还是很大的。同时，身为人工智能方向的学生，通过这个作业让我对自己专业内的知识有了更深入的了解，也对深度学习的流程有了更清晰的概念，而不是只有原来的泛泛的理论。有一点遗憾的是，没能更好的规划代码的结构，一开始参考的深度学习的代码都是多文件项目，代码量较大，让我有点退却。后面自己编写的时候还是保守的只写一个源文件来实现，但编写的过程中也感受到，比如对于预处理的代码应该单独列出等，现在只是简单的将跑过一遍已经存储好了的无需再跑第二遍的代码注释掉，还是略显杂乱。好在代码量本身不大，300多行，就算放在一个文件里也不显得很繁多。不过现在完成了全部之后再来看，

当初的退却更多的是因为自己对框架、流程的不了解，下次再遇到类似的项目，可以进行更多的尝试，更好的规划框架。