在[《文本情感分类（一）：传统模型》](http://spaces.ac.cn/archives/3360/)一文中，笔者简单介绍了进行文本情感分类的传统思路。传统的思路简单易懂，而且稳定性也比较强，然而存在着两个难以克服的局限性：**一、精度问题**，传统思路差强人意，当然一般的应用已经足够了，但是要进一步提高精度，却缺乏比较好的方法；**二、背景知识问题**，传统思路需要事先提取好情感词典，而这一步骤，往往需要人工操作才能保证准确率，换句话说，做这个事情的人，不仅仅要是数据挖掘专家，还需要语言学家，这个背景知识依赖性问题会阻碍着自然语言处理的进步。

庆幸的是，深度学习解决了这个问题（至少很大程度上解决了），它允许我们在几乎“零背景”的前提下，为某个领域的实际问题建立模型。本文延续上一篇文章所谈及的文本情感分类为例，简单讲解深度学习模型。其中上一篇文章已经详细讨论过的部分，本文不再详细展开。

**深度学习与自然语言处理**

近年来，深度学习算法被应用到了自然语言处理领域，获得了比传统模型更优秀的成果。如Bengio等学者基于深度学习的思想构建了神经概率语言模型，并进一步利用各种深层神经网络在大规模英文语料上进行语言模型的训练，得到了较好的语义表征，完成了句法分析和情感分类等常见的自然语言处理任务，为大数据时代的自然语言处理提供了新的思路。

经过笔者的测试，基于深度神经网络的情感分析模型，**其准确率往往有95%以上**，深度学习算法的魅力和威力可见一斑！

关于深度学习进一步的资料，请参考以下文献：

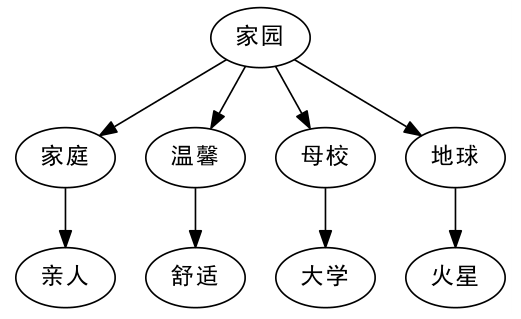
[1] Yoshua Bengio, Réjean Ducharme Pascal Vincent, Christian Jauvin. A Neural Probabilistic Language Model, 2003  
[2] 一种新的语言模型：<http://blog.sciencenet.cn/blog-795431-647334.html>  
[3] Deep Learning（深度学习）学习笔记整理：<http://blog.csdn.net/zouxy09/article/details/8775360>  
[4] Deep Learning：[http://deeplearning.net](http://deeplearning.net/)  
[5] 漫话中文自动分词和语义识别：<http://www.matrix67.com/blog/archives/4212>  
[6] Deep Learning 在中文分词和词性标注任务中的应用：<http://blog.csdn.net/itplus/article/details/13616045>

**语言的表达**

在文章[《闲聊：神经网络与深度学习》](http://spaces.ac.cn/archives/3331/)中，笔者已经提到过，建模环节中最重要的一步是特征提取，在自然语言处理中也不例外。在自然语言处理中，最核心的一个问题是，如何把一个句子用数字的形式有效地表达出来？如果能够完成这一步，句子的分类就不成问题了。显然，一个最初等的思路是：给每个词语赋予唯一的编号1,2,3,4...，然后把句子看成是编号的集合，比如假设1,2,3,4分别代表“我”、“你”、“爱”、“恨”，那么“我爱你”就是[1, 3, 2]，“我恨你”就是[1, 4, 2]。这种思路看起来有效，实际上非常有问题，比如一个稳定的模型会认为3跟4是很接近的，因此[1, 3, 2]和[1, 4, 2]应当给出接近的分类结果，但是按照我们的编号，3跟4所代表的词语意思完全相反，分类结果不可能相同。因此，这种编码方式不可能给出好的结果。

读者也许会想到，我**将意思相近的词语的编号凑在一堆（给予相近的编号）不就行了**？嗯，确实如果，如果有办法把相近的词语编号放在一起，那么确实会大大提高模型的准确率。可是问题来了，如果给出每个词语唯一的编号，并且将相近的词语编号设为相近，实际上是假设了语义的单一性，也就是说，语义仅仅是一维的。然而事实并非如此，语义应该是多维的。

**比如我们谈到“家园”，有的人会想到近义词“家庭”，从“家庭”又会想到“亲人”，这些都是有相近意思的词语；另外，从“家园”，有的人会想到“地球”，从“地球”又会想到“火星”。换句话说，“亲人”、“火星”都可以看作是“家园”的二级近似，但是“亲人”跟“火星”本身就没有什么明显的联系了。此外，从语义上来讲，“大学”、“舒适”也可以看做是“家园”的二级近似，显然，如果仅通过一个唯一的编号，是很难把这些词语放到适合的位置的。**

[](http://spaces.ac.cn/usr/uploads/2015/08/1893427039.png)

**Word2Vec：高维来了**

从上面的讨论可以知道，很多词语的意思是各个方向发散开的，而不是单纯的一个方向，因此唯一的编号不是特别理想。那么，多个编号如何？换句话说，将词语对应一个多维向量？不错，这正是非常正确的思路。

为什么多维向量可行？首先，多维向量解决了词语的多方向发散问题，仅仅是二维向量就可以360度全方位旋转了，何况是更高维呢（实际应用中一般是几百维）。其次，还有一个比较实际的问题，就是多维向量允许我们用变化较小的数字来表征词语。怎么说？我们知道，就中文而言，词语的数量就多达数十万，如果给每个词语唯一的编号，那么编号就是从1到几十万变化，变化幅度如此之大，模型的稳定性是很难保证的。如果是高维向量，比如说20维，那么仅需要0和1就可以表达220=1048576（100万）个词语了。变化较小则能够保证模型的稳定性。

扯了这么多，还没有真正谈到点子上。现在思路是有了，问题是，如何把这些词语放到正确的高维向量中？而且重点是，要在没有语言背景的情况下做到这件事情？（换句话说，如果我想处理英语语言任务，并不需要先学好英语，而是只需要大量收集英语文章，这该多么方便呀！）在这里我们不可能也不必要进行更多的原理上的展开，而是要介绍：而基于这个思路，有一个Google开源的著名的工具——[Word2Vec](https://code.google.com/p/word2vec/)。

简单来说，Word2Vec就是完成了上面所说的我们想要做的事情——用高维向量（词向量，Word Embedding）表示词语，并把相近意思的词语放在相近的位置，而且用的是实数向量（不局限于整数）。我们只需要有大量的某语言的语料，就可以用它来训练模型，获得词向量。词向量好处前面已经提到过一些，或者说，它就是问了解决前面所提到的问题而产生的。另外的一些好处是：词向量可以方便做聚类，用欧氏距离或余弦相似度都可以找出两个具有相近意思的词语。这就相当于解决了“一义多词”的问题（遗憾的是，似乎没什么好思路可以解决一词多义的问题。）

关于Word2Vec的数学原理，读者可以参考[这系列文章](http://blog.csdn.net/itplus/article/details/37969519)。而Word2Vec的实现，Google官方提供了C语言的源代码，读者可以自行编译。而Python的Gensim库中也提供现成的Word2Vec作为子库（事实上，这个版本貌似比官方的版本更加强大）。

**表达句子：句向量**

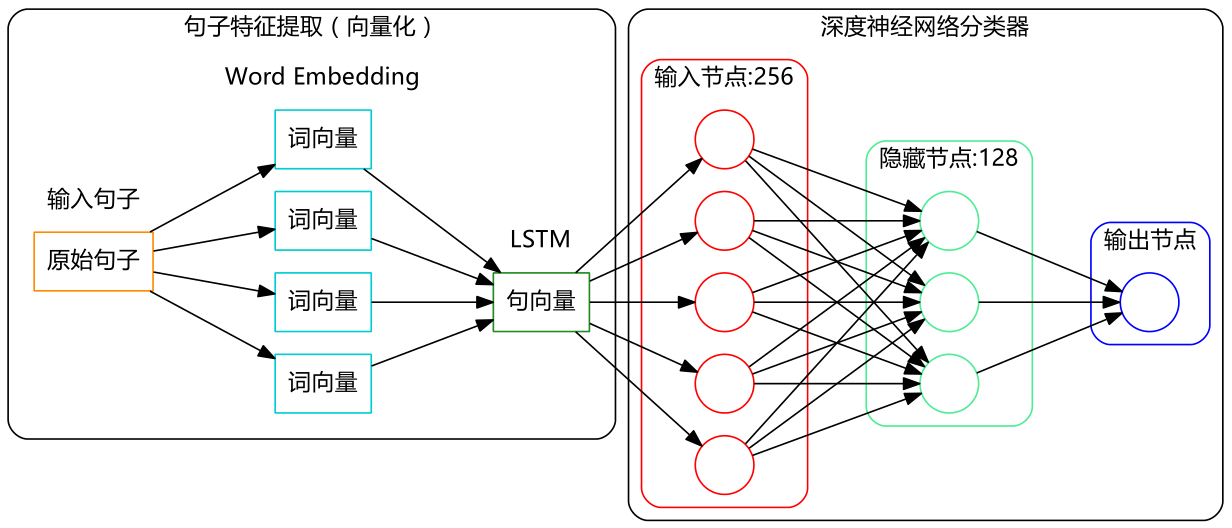
接下来要解决的问题是：我们已经分好词，并且已经将词语转换为高维向量，那么句子就对应着词向量的集合，也就是矩阵，类似于图像处理，图像数字化后也对应一个像素矩阵；可是模型的输入一般只接受一维的特征，那怎么办呢？一个比较简单的想法是将矩阵展平，也就是将词向量一个接一个，组成一个更长的向量。这个思路是可以，但是这样就会使得我们的输入维度高达几千维甚至几万维，事实上是难以实现的。（如果说几万维对于今天的计算机来说不是问题的话，那么对于1000x1000的图像，就是高达100万维了！）

事实上，对于图像处理来说，已经有一套成熟的方法了，叫做**卷积神经网络**（CNNs），它是神经网络的一种，专门用来处理矩阵输入的任务，能够将矩阵形式的输入编码为较低维度的一维向量，而保留大多数有用信息。卷积神经网络那一套也可以直接搬到自然语言处理中，尤其是文本情感分类中，效果也不错，相关的文章有[《Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts》](http://www.aclweb.org/anthology/C14-1008)。但是句子的原理不同于图像，直接将图像那一套用于语言，虽然略有小成，但总让人感觉不伦不类。因此，这并非自然语言处理中的主流方法。

在自然语言处理中，通常用到的方法是**递归神经网络**或**循环神经网络**（都叫RNNs）。它们的作用跟卷积神经网络是一样的，将矩阵形式的输入编码为较低维度的一维向量，而保留大多数有用信息。跟卷积神经网络的区别在于，卷积神经网络更注重全局的模糊感知（好比我们看一幅照片，事实上并没有看清楚某个像素，而只是整体地把握图片内容），而RNNs则是注重邻近位置的重构，由此可见，对于语言任务，RNNs更具有说服力（语言总是由相邻的字构成词，相邻的词构成短语，相邻的短语构成句子，等等，因此，需要有效地把邻近位置的信息进行有效的整合，或者叫重构）。

说到模型的分类，可真谓无穷无尽。在RNNs这个子集之下，又有很多个变种，如普通的RNNs，以及GRU、LSTM等，读者可以参考Keras的官方文档：<http://keras.io/models/>，它是Python是一个深度学习库，提供了大量的深度学习模型，它的官方文档既是一个帮助教程，也是一个模型的列表——它基本实现了目前流行的深度学习模型。

**搭建LSTM模型**

吹了那么久水，是该干点实事了。现在我们基于LSTM（Long-Short Term Memory，长短期记忆人工神经网络）搭建一个文本情感分类的深度学习模型，其结构图如下：  
[](http://spaces.ac.cn/usr/uploads/2015/08/2067741257.png)  
模型结构很简单，没什么复杂的，实现也很容易，用的就是Keras，它都为我们实现好了现成的算法了。

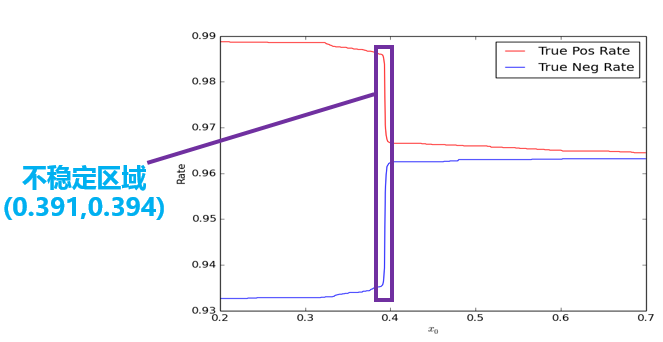
首先将每个单词w\_{m-n+1},w\_{m-n+2} … w\_{m-1}映射到词向量空间，再把各个单词的词向量组合成一个更大的向量作为神经网络输入，输出是P(w\_m)。

现在我们来谈谈有意思的两步。

**第一步是标注语料的收集。**要注意我们的模型是监督训练的（至少也是半监督），所以需要收集一些已经分好类的句子，数量嘛，当然越多越好。而对于中文文本情感分类来说，这一步着实不容易，中文的资料往往是相当匮乏的。笔者在做模型的时候，东拼西凑，通过各种渠道（有在网上搜索下载的、有在数据堂花钱购买的）收集了两万多条中文标注语料（涉及六个领域）用来训练模型。（文末有共享）

[](http://spaces.ac.cn/usr/uploads/2015/08/1709515700.png)

**第二步是模型阈值选取问题。**事实上，训练的预测结果是一个[0, 1]区间的连续的实数，而程序默认情况下会将0.5设为阈值，也就是将大于0.5的结果判断为正，将小于0.5的结果判断为负。这样的默认值在很多情况下并不是最好的。如下图所示，我们在研究不同的阈值对真正率和真负率的影响之时，发现在(0.391, 0.394)区间内曲线曲线了陡变。

[](http://spaces.ac.cn/usr/uploads/2015/08/4127095012.png)

虽然从绝对值看，只是从0.99下降到了0.97，变化不大，但是其变化率是非常大的。正常来说都是平稳变化的，陡变意味着肯定出现了什么异常情况，而显然这个异常的原因我们很难发现。换句话说，这里存在一个不稳定的区域，这个区域内的预测结果事实上是不可信的，因此，保险起见，我们扔掉这个区间。只有结果大于0.394的，我们才认为是正，小于0.391的，我们才认为是负，是0.391到0.394之间的，我们待定。实验表明这个做法有助于提高模型的应用准确率。

**说点总结**

文章很长，粗略地介绍了深度学习在文本情感分类中的思路和实际应用，很多东西都是泛泛而谈。笔者并非要写关于深度学习的教程，而是只想把关键的地方指出来，至少是那些我认为是比较关键的地方。关于深度学习，有很多不错的教程，最好还是阅读英文的论文，中文的比较好的就是博客<http://blog.csdn.net/itplus>了，笔者就不在这方面献丑了。

下面是我的语料和代码。读者可能会好奇我为什么会把这些“私人珍藏”共享呢？其实很简单，因为我不是干这行的哈，数据挖掘对我来说只是一个爱好，一个数学与Python结合的爱好，因此在这方面，我不用担心别人比我领先哈。

语料下载：[sentiment.zip](http://spaces.ac.cn/usr/uploads/2015/08/646864264.zip)  
采集到的评论数据：[sum.zip](http://spaces.ac.cn/usr/uploads/2015/09/829078856.zip)

搭建LSTM做文本情感分类的代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71 | **import** pandas **as** pd *#导入Pandas*  **import** numpy **as** np *#导入Numpy*  **import** jieba *#导入结巴分词*    **from** keras.preprocessing **import** sequence  **from** keras.optimizers **import** SGD, RMSprop, Adagrad  **from** keras.utils **import** np\_utils  **from** keras.models **import** Sequential  **from** keras.layers.core **import** Dense, Dropout, Activation  **from** keras.layers.embeddings **import** Embedding  **from** keras.layers.recurrent **import** LSTM, GRU    **from** \_\_future\_\_ **import** absolute\_import *#导入3.x的特征函数*  **from** \_\_future\_\_ **import** print\_function    neg=pd.read\_excel('neg.xls',header=None,index=None)  pos=pd.read\_excel('pos.xls',header=None,index=None) *#读取训练语料完毕*  pos['mark']=1  neg['mark']=0 *#给训练语料贴上标签*  pn=pd.concat([pos,neg],ignore\_index=True) *#合并语料*  neglen=len(neg)  poslen=len(pos) *#计算语料数目*    cw = **lambda** x: list(jieba.cut(x)) *#定义分词函数*  pn['words'] = pn[0].apply(cw)    comment = pd.read\_excel('sum.xls') *#读入评论内容*  *#comment = pd.read\_csv('a.csv', encoding='utf-8')*  comment = comment[comment['rateContent'].notnull()] *#仅读取非空评论*  comment['words'] = comment['rateContent'].apply(cw) *#评论分词*    d2v\_train = pd.concat([pn['words'], comment['words']], ignore\_index = True)    w = [] *#将所有词语整合在一起*  **for** i **in** d2v\_train:  w.extend(i)    dict = pd.DataFrame(pd.Series(w).value\_counts()) *#统计词的出现次数*  **del** w,d2v\_train  dict['id']=list(range(1,len(dict)+1))    get\_sent = **lambda** x: list(dict['id'][x])  pn['sent'] = pn['words'].apply(get\_sent) *#速度太慢*    maxlen = 50    **print**("Pad sequences (samples x time)")  pn['sent'] = list(sequence.pad\_sequences(pn['sent'], maxlen=maxlen))    x = np.array(list(pn['sent']))[::2] *#训练集*  y = np.array(list(pn['mark']))[::2]  xt = np.array(list(pn['sent']))[1::2] *#测试集*  yt = np.array(list(pn['mark']))[1::2]  xa = np.array(list(pn['sent'])) *#全集*  ya = np.array(list(pn['mark']))    **print**('Build model...')  model = Sequential()  model.add(Embedding(len(dict)+1, 256))  model.add(LSTM(256, 128)) *# try using a GRU instead, for fun*  model.add(Dropout(0.5))  model.add(Dense(128, 1))  model.add(Activation('sigmoid'))    model.compile(loss='binary\_crossentropy', optimizer='adam', class\_mode="binary")    model.fit(xa, ya, batch\_size=16, nb\_epoch=10) *#训练时间为若干个小时*    classes = model.predict\_classes(xa)  acc = np\_utils.accuracy(classes, ya)  **print**('Test accuracy:', acc) |

[首页](http://spaces.ac.cn/) [信息时代](http://spaces.ac.cn/category/Big-Data/) 文本情感分类（三）：分词 OR 不分词

**29Jun**

# [文本情感分类（三）：分词 OR 不分词](http://spaces.ac.cn/archives/3863/)

作者：苏剑林 | 发布时间：2016-06-29

**本文目录 [**[**隐藏**](javascript:content_index_toggleToc())**]**

* [模型测试](http://spaces.ac.cn/archives/3863/#模型测试)
* [多扯一点](http://spaces.ac.cn/archives/3863/#多扯一点)
* [代码来了](http://spaces.ac.cn/archives/3863/#代码来了)

去年泰迪杯竞赛过后，笔者写了一篇简要介绍深度学习在情感分析中的应用的博文[《文本情感分类（二）：深度学习模型》](http://spaces.ac.cn/archives/3414/)。虽然文章很粗糙，但还是得到了不少读者的反响，让我颇为意外。然而，那篇文章中在实现上有些不清楚的地方，这是因为：1、在那篇文章以后，keras已经做了比较大的改动，原来的代码不通用了；2、里边的代码可能经过我随手改动过，所以发出来的时候不是最适当的版本。因此，在近一年之后，我再重拾这个话题，并且完成一些之前没有完成的测试。

为什么要用深度学习模型？除了它更高精度等原因之外，还有一个重要原因，那就是它是**目前唯一的能够实现“端到端”的模型**。所谓“端到端”，就是能够直接将原始数据和标签输入，然后让模型自己完成一切过程——包括特征的提取、模型的学习。而回顾我们做中文情感分类的过程，一般都是“分词——词向量——句向量(LSTM)——分类”这么几个步骤。虽然很多时候这种模型已经达到了state of art的效果，但是有些疑问还是需要进一步测试解决的。对于中文来说，字才是最低粒度的文字单位，因此从“端到端”的角度来看，应该将直接将句子以字的方式进行输入，而不是先将句子分好词。**那到底有没有分词的必要性呢？本文测试比较了字one hot、字向量、词向量三者之间的效果。**

### 模型测试

本文测试了三个模型，或者说，是三套框架，具体代码在文末给出。这三套框架分别是：

1、**one hot**：以字为单位，不分词，将每个句子截断为200字（不够则补空字符串），然后将句子以“字-one hot”的矩阵形式输入到LSTM模型中进行学习分类；

2、**one embedding**：以字为单位，不分词，，将每个句子截断为200字（不够则补空字符串），然后将句子以“字-字向量(embedding)“的矩阵形式输入到LSTM模型中进行学习分类；

3、**word embedding**：以词为单位，分词，，将每个句子截断为100词（不够则补空字符串），然后将句子以“词-词向量(embedding)”的矩阵形式输入到LSTM模型中进行学习分类。

其中所用的LSTM模型结构是类似的。所用的语料还是[《文本情感分类（二）：深度学习模型》](http://spaces.ac.cn/archives/3414/)中的语料，以15000条进行训练，剩下的6000条左右做测试。意外的是，三个模型都取得了相近的结果。

迭代次数每轮用时训练准确率测试准确率one hot90100s96.60%89.21%one embedding3036s95.95%89.55%word embedding3018s98.41%89.03%one hotone embeddingword embedding迭代次数903030每轮用时100s36s18s训练准确率96.60%95.95%98.41%测试准确率89.21%89.55%89.03%

可见，在准确率方面，三者是类似的，区分度不大。不管是用one hot、字向量还是词向量，结果都差不多。也许用[《文本情感分类（二）：深度学习模型》](http://spaces.ac.cn/archives/3414/)的方法来为每个模型选取适当的阈值，会使得测试准确率更高一些，但模型之间的相对准确率应该不会变化很大。

当然，测试本身可能存在一些不公平的情况，也许会导致测试结果公平，而我也没有反复去测试。比如one hot的模型迭代了90次，其它两个模型是30次，因为one hot模型所构造的样本维度太大，需要经过更长时间才出现收敛现象，而且训练过程中，准确率是波动上升的，并非像其它两个模型那样稳定上升。事实上这是所有one hot模型的共同特点。

### 多扯一点

看上去，one hot模型的确存在维度灾难的问题，而且训练时间又长，效果又没有明显提升，那是否就说明没有研究one hot表示的必要了呢？

我觉得不是这样的。当初大家诟病one hot模型的原因，除了维度灾难之外，还有一个就是“语义鸿沟”，也就说任意两个词之间没有任何相关性（不管用欧式距离还是余弦相似度，任意两个词的计算结果是一样的）。可是，这一点假设用在词语中不成立，可是用在中文的“字”上面，不是很合理吗？汉字单独成词的例子不多，大多数是二字词，也就是说，任意两个字之间没有任何相关性，这个假设在汉字的“字”的层面上，是近似成立的！而后面我们用了LSTM，LSTM本身具有整合邻近数据的功能，因此，它暗含了将字整合为词的过程。

此外，one hot模型还有一个非常重要的特点——它**没有任何信息损失**——从one hot的编码结果中，我们反过来解码出原来那句话是哪些字词组成的，然而，我无法从一个词向量中确定原来的词是什么。这些观点都表明，在很多情况下，one hot模型都是很有价值的。

而我们为什么用词向量呢？词向量相当于做了一个假设：每个词具有比较确定的意思。这个假设在词语层面也是近似成立的，毕竟一词多义的词语相对来说也不多。正因为如此，我们才可以将词放到一个较低维度的实数空间里，用一个实数向量来表示一个词语，并且用它们之间的距离或者余弦相似度来表示词语之间的相似度。这也是词向量能够解决“一义多词”而没法解决“一词多义”的原因。

从这样看来，上面三个模型中，只有one hot和word embedding才是理论上说得过去的，而one embedding则看上去变得不伦不类了，因为字似乎不能说具有比较确定的意思。但为什么one embedding效果也还不错？我估计，这可能是因为二元分类问题本身是一个很粗糙的分类（0或1），如果更多元的分类，可能one embedding的方式效果就降下来了。不过，我也没有进行更多的测试了，因为太耗时间了。

当然，这只能算是我的**主观臆测**，还望大家指正。尤其是one embedding部分的评价，是值得商榷的。

### 代码来了

可能大家并不想看我胡扯一通，是直接来看代码的，现奉上三个模型的代码。最好有GPU加速，尤其是试验one hot模型，不然慢到哭了。

模型1：one hot

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90 | *# -\*- coding:utf-8 -\*-*    '''  one hot测试  在GTX960上，约100s一轮  经过90轮迭代，训练集准确率为96.60%，测试集准确率为89.21%  Dropout不能用太多，否则信息损失太严重  '''    **import** numpy **as** np  **import** pandas **as** pd    pos = pd.read\_excel('pos.xls', header=None)  pos['label'] = 1  neg = pd.read\_excel('neg.xls', header=None)  neg['label'] = 0  all\_ = pos.append(neg, ignore\_index=True)    maxlen = 200 *#截断字数*  min\_count = 20 *#出现次数少于该值的字扔掉。这是最简单的降维方法*    content = ''.join(all\_[0])  abc = pd.Series(list(content)).value\_counts()  abc = abc[abc >= min\_count]  abc[:] = range(len(abc))    **def** doc2num(s, maxlen):  s = [i **for** i **in** s **if** i **in** abc.index]  s = s[:maxlen]  **return** list(abc[s])    all\_['doc2num'] = all\_[0].apply(**lambda** s: doc2num(s, maxlen))    *#手动打乱数据*  *#当然也可以把这部分加入到生成器中*  idx = range(len(all\_))  np.random.shuffle(idx)  all\_ = all\_.loc[idx]    *#按keras的输入要求来生成数据*  x = np.array(list(all\_['doc2num']))  y = np.array(list(all\_['label']))  y = y.reshape((-1,1)) *#调整标签形状*      **from** keras.utils **import** np\_utils  **from** keras.models **import** Sequential  **from** keras.layers **import** Dense, Activation, Dropout  **from** keras.layers **import** LSTM  **import** sys  sys.setrecursionlimit(10000) *#增大堆栈最大深度(递归深度)，据说默认为1000，报错*    *#建立模型*  model = Sequential()  model.add(LSTM(128, input\_shape=(maxlen,len(abc))))  model.add(Dropout(0.5))  model.add(Dense(1))  model.add(Activation('sigmoid'))  model.compile(loss='binary\_crossentropy',  optimizer='rmsprop',  metrics=['accuracy'])      *#单个one hot矩阵的大小是maxlen\*len(abc)的，非常消耗内存*  *#为了方便低内存的PC进行测试，这里使用了生成器的方式来生成one hot矩阵*  *#仅在调用时才生成one hot矩阵*  *#可以通过减少batch\_size来降低内存使用，但会相应地增加一定的训练时间*  batch\_size = 128  train\_num = 15000    *#不足则补全0行*  gen\_matrix = **lambda** z: np.vstack((np\_utils.to\_categorical(z, len(abc)), np.zeros((maxlen-len(z), len(abc)))))    **def** data\_generator(data, labels, batch\_size):  batches = [range(batch\_size\*i, min(len(data), batch\_size\*(i+1))) **for** i **in** range(len(data)/batch\_size+1)]  **while** True:  **for** i **in** batches:  xx = np.zeros((maxlen, len(abc)))  xx, yy = np.array(map(gen\_matrix, data[i])), labels[i]  **yield** (xx, yy)      model.fit\_generator(data\_generator(x[:train\_num], y[:train\_num], batch\_size), samples\_per\_epoch=train\_num, nb\_epoch=30)    model.evaluate\_generator(data\_generator(x[train\_num:], y[train\_num:], batch\_size), val\_samples=len(x[train\_num:]))    **def** predict\_one(s): *#单个句子的预测函数*  s = gen\_matrix(doc2num(s, maxlen))  s = s.reshape((1, s.shape[0], s.shape[1]))  **return** model.predict\_classes(s, verbose=0)[0][0] |

模型2：one embedding

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72 | *# -\*- coding:utf-8 -\*-*    '''  one embedding测试  在GTX960上，36s一轮  经过30轮迭代，训练集准确率为95.95%，测试集准确率为89.55%  Dropout不能用太多，否则信息损失太严重  '''      **import** numpy **as** np  **import** pandas **as** pd    pos = pd.read\_excel('pos.xls', header=None)  pos['label'] = 1  neg = pd.read\_excel('neg.xls', header=None)  neg['label'] = 0  all\_ = pos.append(neg, ignore\_index=True)    maxlen = 200 *#截断字数*  min\_count = 20 *#出现次数少于该值的字扔掉。这是最简单的降维方法*    content = ''.join(all\_[0])  abc = pd.Series(list(content)).value\_counts()  abc = abc[abc >= min\_count]  abc[:] = range(1, len(abc)+1)  abc[''] = 0 *#添加空字符串用来补全*    **def** doc2num(s, maxlen):  s = [i **for** i **in** s **if** i **in** abc.index]  s = s[:maxlen] + ['']\*max(0, maxlen-len(s))  **return** list(abc[s])    all\_['doc2num'] = all\_[0].apply(**lambda** s: doc2num(s, maxlen))    *#手动打乱数据*  idx = range(len(all\_))  np.random.shuffle(idx)  all\_ = all\_.loc[idx]    *#按keras的输入要求来生成数据*  x = np.array(list(all\_['doc2num']))  y = np.array(list(all\_['label']))  y = y.reshape((-1,1)) *#调整标签形状*      **from** keras.models **import** Sequential  **from** keras.layers **import** Dense, Activation, Dropout, Embedding  **from** keras.layers **import** LSTM    *#建立模型*  model = Sequential()  model.add(Embedding(len(abc), 256, input\_length=maxlen))  model.add(LSTM(128))  model.add(Dropout(0.5))  model.add(Dense(1))  model.add(Activation('sigmoid'))  model.compile(loss='binary\_crossentropy',  optimizer='adam',  metrics=['accuracy'])    batch\_size = 128  train\_num = 15000    model.fit(x[:train\_num], y[:train\_num], batch\_size = batch\_size, nb\_epoch=30)    model.evaluate(x[train\_num:], y[train\_num:], batch\_size = batch\_size)    **def** predict\_one(s): *#单个句子的预测函数*  s = np.array(doc2num(s, maxlen))  s = s.reshape((1, s.shape[0]))  **return** model.predict\_classes(s, verbose=0)[0][0] |

模型3：word embedding

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | *# -\*- coding:utf-8 -\*-*    '''  word embedding测试  在GTX960上，18s一轮  经过30轮迭代，训练集准确率为98.41%，测试集准确率为89.03%  Dropout不能用太多，否则信息损失太严重  '''    **import** numpy **as** np  **import** pandas **as** pd  **import** jieba    pos = pd.read\_excel('pos.xls', header=None)  pos['label'] = 1  neg = pd.read\_excel('neg.xls', header=None)  neg['label'] = 0  all\_ = pos.append(neg, ignore\_index=True)  all\_['words'] = all\_[0].apply(**lambda** s: list(jieba.cut(s))) *#调用结巴分词*    maxlen = 100 *#截断词数*  min\_count = 5 *#出现次数少于该值的词扔掉。这是最简单的降维方法*    content = []  **for** i **in** all\_['words']:  content.extend(i)    abc = pd.Series(content).value\_counts()  abc = abc[abc >= min\_count]  abc[:] = range(1, len(abc)+1)  abc[''] = 0 *#添加空字符串用来补全*    **def** doc2num(s, maxlen):  s = [i **for** i **in** s **if** i **in** abc.index]  s = s[:maxlen] + ['']\*max(0, maxlen-len(s))  **return** list(abc[s])    all\_['doc2num'] = all\_['words'].apply(**lambda** s: doc2num(s, maxlen))    *#手动打乱数据*  idx = range(len(all\_))  np.random.shuffle(idx)  all\_ = all\_.loc[idx]    *#按keras的输入要求来生成数据*  x = np.array(list(all\_['doc2num']))  y = np.array(list(all\_['label']))  y = y.reshape((-1,1)) *#调整标签形状*      **from** keras.models **import** Sequential  **from** keras.layers **import** Dense, Activation, Dropout, Embedding  **from** keras.layers **import** LSTM    *#建立模型*  model = Sequential()  model.add(Embedding(len(abc), 256, input\_length=maxlen))  model.add(LSTM(128))  model.add(Dropout(0.5))  model.add(Dense(1))  model.add(Activation('sigmoid'))  model.compile(loss='binary\_crossentropy',  optimizer='adam',  metrics=['accuracy'])    batch\_size = 128  train\_num = 15000    model.fit(x[:train\_num], y[:train\_num], batch\_size = batch\_size, nb\_epoch=30)    model.evaluate(x[train\_num:], y[train\_num:], batch\_size = batch\_size)    **def** predict\_one(s): *#单个句子的预测函数*  s = np.array(doc2num(list(jieba.cut(s)), maxlen))  s = s.reshape((1, s.shape[0]))  **return** model.predict\_classes(s, verbose=0)[0][0] |