6 【中文分词系列】7. 深度学习分词？只需一个词典！

Mar By 苏剑林 | 2017-03-06 | 47661位读者 引用

这个系列慢慢写到第7篇，基本上也把分词的各种模型理清楚了，除了一些细微的调整（比如最后的分类器换成CRF）外，剩下的就看怎么玩了。基本上来说，要速度，就用基于词典的分词，要较好地解决组合歧义何和新词识别，则用复杂模型，比如之前介绍的LSTM、FCN都可以。但问题是，用深度学习训练分词器，需要标注语料，这费时费力，仅有的公开的几个标注语料，又不可能赶得上时效，比如，几乎没有哪几个公开的分词系统能够正确切分出“扫描二维码，关注微信号”来。

本文就是做了这样的一个实验，仅用一个词典，就完成了一个深度学习分词器的训练，居然效果还不错！这种方案可以称得上是半监督的，甚至是无监督的。

随机组合就可以

做法很简单，既然深度学习需要语料，那我就自己生成一批语料。怎么生成？我把词典中的词随机组合就行了。不对不对，随机组合生成的不是自然语言呀？我开始也怀疑这一点，但实验之后发现，这样做出来的效果特别好，甚至有胜于标注语料的结果的现象。

事不宜迟，我们来动手。首先得准备一个带词频的词表，词频是必须的，如果没有词频，则效果会大打折扣。然后写一个函数，以正比于词频的概率，随机挑选词表中的词语，组合成“句子”。

```
import numpy as np
import pandas as pd

class Random_Choice:
    def __init__(self, elements, weights):
        d = pd.DataFrame(zip(elements, weights))
        self.elements, self.weights = [], []
        for i,j in d.groupby(1):
            self.weights.append(len(j)*i)
            self.elements.append(tuple(j[0]))
        self.weights = np.cumsum(self.weights).astype(np.float64)/sum(self.weights)
    def choice(self):
        r = np.random.random()
        w = self.elements[np.where(self.weights >= r)[0][0]]
        return w[np.random.randint(0, len(w))]
```

注意，这里按了权重（weights）来分组，从而实现了随机抽。随机抽的速度取决于分组的数目，所以，词典的词频最好可以做一些预处理，使得它有“扎堆”的现象，即可以把出现10001,10002,10003次的词语都可以取整为10000，把出现12001,12002,12003,12004次的词语都取整为12000，类似这样的预处理。这步是相当重要的，因为如果有GPU的话，后面模型的训练速度瓶颈就是在这里了。

接着，统计字表，然后写生成器，这些都是很常规的，用的方法还是4tag字标注法，然后添加x标签代表填充标注，不清楚的读者，可以回头阅读LSTM分词的文章：

```
import pickle
words = pd.read_csv('dict.txt', delimiter='\t', header=None, encoding='utf-8')
words[0] = words[0].apply(unicode)
```

```
words = words.set_index(0)[1]

try:
    char2id = pickle.load(open('char2id.dic'))
except:
    from collections import defaultdict
    print u'fail to load old char2id.'
    char2id = pd.Series(list(''.join(words.index))).value_counts()
    char2id[:] = range(1, len(char2id)+1)
    char2id = defaultdict(int, char2id.to_dict())
    pickle.dump(char2id, open('char2id.dic', 'w'))

word_size = 128
maxlen = 48
batch_size = 1024

def word2tag(s):
    if len(s) == 1:
        return 's'
    elif len(s) >= 2:
        return 'b'+'m'*(len(s)-2)+'e'

tag2id = {'s':[1,0,0,0,0], 'b':[0,1,0,0,0], 'm':[0,0,1,0,0], 'e':[0,0,0,1,0]}

def data_generator():
    wc = Random_Choice(words.index, words)
    x, y = [], []
    while True:
        n = np.random.randint(1, 17)
        seq = [wc.choice() for i in range(n)]
        tag = ''.join([word2tag(i) for i in seq])
        seq = [char2id[i] for i in ''.join(seq)]
        if len(seq) > maxlen:
            continue
        else:
            seq = seq + [0]*(maxlen-len(seq))
            tag = [tag2id[i] for i in tag]
            tag = tag + [[0,0,0,0,1]]*(maxlen-len(tag))
            x.append(seq)
            y.append(tag)
        if len(x) == batch_size:
            yield np.array(x), np.array(y)
            x, y = [], []
```

还是以前的模型

模型的话，用回以前写过的LSTM或者CNN都行，我这里用了LSTM，结果表明LSTM具有很强的记忆能力。

```
# Keras 2.0 + Tensorflow 1.0 运行通过

from keras.layers import Dense, Embedding, LSTM, TimeDistributed, Input, Bidirectional
from keras.models import Model

sequence = Input(shape=(maxlen,), dtype='int32')
```

```
embedded = Embedding(len(char2id)+1, word_size, input_length=maxlen, mask_zero=True)(se
blstm = Bidirectional(LSTM(64, return_sequences=True))(embedded)
output = TimeDistributed(Dense(5, activation='softmax'))(blstm)
model = Model(inputs=sequence, outputs=output)
model.compile(loss='categorical_crossentropy', optimizer='adam')

try:
    model.load_weights('model.weights')
except:
    print u'fail to load old weights.'

for i in range(100):
    print i
    model.fit_generator(data_generator(), steps_per_epoch=100, epochs=10)
    model.save_weights('model.weights')
```

用我的GTX1060，结合我的词典（50万不同词语），每轮大概要花70s，这里每10轮保存一次模型，而range(100)这个100是随便写的，因为每隔10轮就保存一次，所以读者可以随时看效果中断程序。

关于准确率的问题，要注意的是因为使用了mask_zero=True，所以x标签被忽略了，训练过程但最后显示的训练准确率，又把x标签算进去了，所以最后现实的准确率只能不到0.3，大概就是0.28左右。这不重要的，我们训练完成后，再测试即可。

最后是一个经验：字向量维度越大，对长词的识别效果一般越好。

结合动态规划输出

到这里，就结合一下viterbi算法，通过动态规划来输出最后的结果。动态规划能够保证输出最优的结果，但是会降低效率。而直接输出分类器预测的最大结果，也能够得到类似的结果（但理论上有可能出现bbbb这样的标注结果）。这个看情况而定吧，这里毕竟是实验，所以就使用了viterbi，如果真的在生产环境中，为了追求速度，应该不用为好（放弃一点精度，大幅提高速度）。

```
zy = {'be':0.5,
      'bm':0.5,
      'eb':0.5,
      'es':0.5,
      'me':0.5,
      'mm':0.5,
      'sb':0.5,
      'ss':0.5
     }

zy = {i:np.log(zy[i]) for i in zy.keys()}

def viterbi(nodes):
    paths = {'b':nodes[0]['b'], 's':nodes[0]['s']}
    for l in range(1,len(nodes)):
        paths_ = paths.copy()
        paths = {}
        for i in nodes[l].keys():
            nows = {}
```

```
            for j in paths_.keys():
                if j[-1]+i in zy.keys():
                    nows[j+i]= paths_[j]+nodes[l][i]+zy[j[-1]+i]
            k = np.argmax(nows.values())
            paths[nows.keys()[k]] = nows.values()[k]
    return paths.keys()[np.argmax(paths.values())]

def simple_cut(s):
    if s:
        s = s[:maxlen]
        r = model.predict(np.array([[char2id[i] for i in s]+[0]*(maxlen-len(s))]), verb
        r = np.log(r)
        nodes = [dict(zip(['s','b','m','e'], i[:4])) for i in r]
        t = viterbi(nodes)
        words = []
        for i in range(len(s)):
            if t[i] in ['s', 'b']:
                words.append(s[i])
            else:
                words[-1] += s[i]
        return words
    else:
        return []

import re
not_cuts = re.compile(u'([\da-zA-Z ]+)|[。，、？！\.\?,!“”]')
def cut_word(s):
    result = []
    j = 0
    for i in not_cuts.finditer(s):
        result.extend(simple_cut(s[j:i.start()]))
        result.append(s[i.start():i.end()])
        j = i.end()
    result.extend(simple_cut(s[j:]))
    return result
```

这里的代码跟之前一样，基本没怎么改。效率并不是很高，不过都说了，这里是实验，如果有兴趣用到生产环境的朋友，自己想办法优化吧。

来测试一下

结合我自己整理的词典，最终的模型在**backoff2005的评测集上达到85%左右的准确率**（backoff2005提供的score脚本算出的准确率），这个准确率取决于你的词典。

看上去很糟糕？这不重要。首先，我们并没有用它的训练集，纯粹使用词典无监督的训练，这个准确率，已经很让人满意了。其次，准确率看上去低，但实际情况更好，因为这不过是语料的分词标准问题而已，比如

> 1、评测集的标准答案，将“古老的中华文化”分为“古老/的/中华/文化”，而模型将它分为“古老/的/中华文化”；
>
> 2、评测集的标准答案，将“在录入上走向中西文求同的道路”分为“在/录入/上/走/向/中/西文/求/同/的/道路”，而模型将它分为“在/录入/上/走向/中西文/求同/的/道路”；

> 3、评测集的标准答案，将“更是教育学家关心的问题”分为“更/是/教育学/家/关心/的/问题”，而模型将它分为“更是/教育学家/关心/的/问题”；

这些例子随便扫一下，还能举出很多，这说明backoff2005的标注本身也不是特别标准，我们不用太在意这个准确率，而更应该留意到，我们得到的模型，对新词、长词，有着更好的识别效果，而达到这个效果，只需要一个词典，这比标注数据容易多了，这对于一些特定领域（比如医学、旅游等等）定制分词系统，是非常有效的。

下面再给一些测试例子，这些例子基本比之前有监督训练的结果还要好：

> 罗斯福 是 第二次世界大战 期间 同盟国 阵营 的 重要 领导人 之一 。 1941 年 珍珠港 事件 发生 后 ， 罗斯福 力主 对 日本 宣战 ， 并 引进 了 价格 管制 和 配给 。 罗斯福 以 租 借 法案 使 美国 转变 为 “ 民主 国家 的 兵工厂 ” ， 使 美国 成为 同盟国 主要 的 军火 供应商 和 融资者 ， 也 使得 美国 国内 产业 大幅 扩张 ， 实现 充分 就业 。 二战 后期 同盟国 逐渐 扭转 形势 后 ， 罗斯福 对 塑造 战后 世界 秩序 发挥 了 关键 作用 ， 其 影响力 在 雅尔塔 会议 及 联合国 的 成立 中 尤其 明显 。 后来 ， 在 美国 协助 下 ， 盟军 击败 德国 、 意大利 和 日本 。
>
> 苏剑林 是 科学 空间 的 博主
>
> 结婚 的 和 尚未 结婚 的
>
> 大肠杆菌 是 人 和 许多 动物 肠道 中 最 主要 且 数量 最 多 的 一种 细菌
>
> 九寨沟 国家级 自然保护区 位于 四川省 阿坝藏族羌族自治州 南坪县 境内 ， 距离 成都市 400 多 公里 ， 是 一条 纵深 40 余公里 的 山沟 谷地
>
> 现代 内地 人 ， 因 莫高窟 而 知 敦煌 。 敦煌 因 莫高窟 也 在 近代 蜚声 海外 。 但 莫高窟 始 凿 于 四 世纪 ， 到 1900 年 才 为 世人 瞩目 ， 而 敦煌 早 从 汉武帝 时 ， 即 公元前 一百多年 起 ， 就 已 是 西北 名城 了
>
> 从前 对 巴特农 神庙 怎么 干 ， 现在 对 圆明园 也 怎么 干 ， 只是 更 彻底 ， 更 漂亮 ， 以至于 荡然无存 。 我们 所有 大 教堂 的 财宝 加 在 一起 ， 也许 还 抵不上 东方 这座 了不起 的 富丽堂皇 的 博物馆 。 那儿 不仅仅 有 艺术 珍品 ， 还有 大堆 的 金银 制品 。 丰功伟绩 ！ 收获 巨大 ！ 两个 胜利者 ， 一个 塞满 了 腰包 ， 这 是 看得见 的 ， 另 一个 装满 了 箱子 。

别忘了，这仅仅是用一个词典做出来的，分出来的不少词，比如尤其是人名，都是词典中没有的。这效果，足可让人满意了吧？

想想为什么

回到我们一开始的疑惑，为什么随机组合的文本也能够训练出一个很棒的分词器出来？原因就在于，我们一开始，基于词典的分词，就是做了个假设：句子是由词随机组合起来的。这样，我们分词，就要对字符串进行切分，使得如下概率：

$$p(w_1)p(w_2)\dots p(w_n)$$

最大。最后求解的过程就用到了动态规划。

而这里，我们不外乎也沿用了这种假设——文本是随机组合的——这个假设严格上来说不成立，但一般来说够用了，效果也显示在这里了。可能让人意外的是，这样出来的分词器，居然也能把“结婚的和尚未结婚的”的组合歧义句分出来。其实不难理解，我们随机组合的时候，是按照词频来挑选词语的，这就导致了高频出现更多，低频出现更少，这样经过大量重复操作后，事实上我们是通过LSTM来学习到了动态规划这个过程！

这是很惊人的，这表示，我们可以用LSTM，来学习传统的一些优化算法！更甚地，通过改进RNN用来解决一些传统cs问题，比如凸包，三角剖分，甚至是TSP，最神奇的地方在于这玩意效果竟然还不错，甚至比一些近似算法效果好。（https://www.zhihu.com/question/47563637）注意，诸如TSP之类的问题，是一个NP问题，原则上没有多项式时间的解法，但利用LSTM，甚至我们有可能得到一个线性的有效解法，这对于传统算法领域是多么大的冲击！用神经网络来设计优化算法，最终用优化算法来优化神经网络，实现一个自己优化自己，那才叫智能！

呃～扯远了，总之，效果是唯一标注吧。

已经训练好的模型

最后分享一个已经训练好的模型，有Keras的读者可以下载测试：

~~seg.zip（该权重最新版本的tensorflow+Keras已经不可用，请大家根据上述代码自行训练～）~~

转载到请包括本文地址：https://kexue.fm/archives/4245

更详细的转载事宜请参考：《科学空间FAQ》

如果您需要引用本文，请参考：

苏剑林. (2017, Mar 06). 《【中文分词系列】7. 深度学习分词？只需一个词典！》[Blog post]. Retrieved from https://kexue.fm/archives/4245