

Report of Deep Learning for Natural Language Processing

高志磊

gaozhilei@buaa.edu.cn

Introduction

从给定的语料库中均匀抽取 200 个段落，将从同一小说中抽取的段落组合到一起，形成 16 篇文档，作为训练集。将 200 个段落作为测试集。利用 LDA 模型进行文本建模，利用训练集训练出不同主题的词频分布，然后将 200 个段落表示为主题分布后进行分类，验证分类结果，并分析主题个数分类性能的变换以及分别以“词”和“字”为基本单元下分类结果的差异。

Methodology

M1: LDA 模型概述

LDA 模型是一种用于文本分析的概率模型，它最早由 Blei 等人在 2003 年提出，旨在通过对文本数据进行分析，自动发现其隐藏的主题结构。被广泛应用于文本挖掘、信息检索、自然语言处理等领域。

LDA 模型的核心思想是将文本表示为一组概率分布，其中每个文档由多个主题混合而成，每个主题又由多个单词组成。LDA 模型的基本原理是先假设一个文本集合的生成过程为：首先从主题分布中随机选择一个主题，然后从该主题的单词分布中随机选择一个单词，重复上述过程，直到生成整个文本。具体来说，LDA 模型的生成过程包括以下三个步骤：

- 1) 对一篇文档的每个位置，从主题分布中抽取一个主题；
- 2) 从上述被抽到的主题所对应的单词分布中抽取一个单词；
- 3) 重复上述过程直至遍历文档中的每一个单词。

M2: LDA 模型生成

首先定义文档 (doc) 集合为 Doc，文章主题 (topic) 集合为 Topic，Doc 中的每个文档 doc 可以看做一个单词序列 $\langle w_1, w_2, \dots, w_n \rangle$ ，其中 w_i 表示第 i 个单词，假设一篇文档 doc 共有 n 个单词。

Doc 中的所有不同单词组成一个集合 Voc，LDA 模型以文档集合 Doc 作为输入，最终训练出两个结果向量， k 表示 Topic 中所有主题的数量， m 表示 Voc 中所有词的数量。

对每个文档 doc，对应到不同 topic 的概率 $\theta_d = \langle p_{t1}, p_{t2}, \dots, p_{tk} \rangle$ ，其中 p_{ti} 表示 doc 对应 Topic 中第 i 个 topic 中词的概率，计算方法如下：

$$p_{ti} = \frac{n_{ti}}{n}$$

n_{ti} 表示 doc 中对应第 i 个 topic 的词数目， n 表示 doc 中所有词的总数。

对 Topic 中的第 t 个 topic，生成不同单词的概率 $\varphi_t = \langle p_{w1}, p_{w2}, \dots, p_{wm} \rangle$ ，其中， p_{wi} 表示第 t 个 topic 生成 Voc 中第 i 个单词的概率。计算方法如下：

$$p_{wi} = \frac{N_{wi}}{N}$$

其中 N_{wi} 表示对应到第 t 个 topic 的 Voc 中第 i 个单词的数目， N 表示所有对应到第 t 个 topic 的单词总数。

LDA 的核心公式如下：

$$p(w|d) = p(w|t)p(t|d)$$

直观的看这个公式，就是以 Topic 作为中间层，可以通过当前的 θ_d 和 φ_t 给出了文档 doc 中出现单词 w 的概率。其中 $p(t|d)$ 利用 θ_d 计算得到， $p(w|t)$ 利用 φ_t 计算得到。实际上，利用当前的 θ_d 和 φ_t ，我们可以为一个文档中的一个单词计算它对应任意一个 topic 时的 $p(w|d)$ ，然后根据这些结果来更新这个词应该对应的 topic。然后，如果这个更新改变了这个单词所对应的 topic，就会反过来影响 θ_d 和 φ_t 。

下面给出一种 LDA 的学习过程：

LDA 算法开始时，先随机地给 θ_d 和 φ_t 赋值（对所有的 doc 和 t ）。然后上述过程不断重复，最终收敛到的结果就是 LDA 的输出。再详细说一下这个迭代的学习过程：

1) 针对一个特定的文档 d_s 中的第 i 个单词 w_i ，如果令该单词对应的 topic 为 t_j ，可以把上述公式改写为：

$$p_j(w_i | d_s) = p(w_i | t_j)p(t_j | d_s) \quad (1.1)$$

2) 现在我们可以枚举 Topic 中的 topic，得到所有的 $p_j(w_i | d_s)$ ，其中 $j = 1, \dots, k$ 。然后可以根据这些概率值结果为 d_s 中的第 i 个单词 w_i 选择一个 topic。最简单的想法是取令 $p_j(w_i | d_s)$ 最大的 t_j （注意，这个式子里只有 j 是变量），即 $\arg \max_j p_j(w_i | d_s)$

3) 然后，如果 d_s 中的第 i 个单词 w_i 在这里选择了一个与原先不同的 topic，就会对 θ_d 和 φ_t 有影响了（根据前面提到过的这两个向量的计算公式可以很容易知道）。它们的影响又会反过来影响对上面提到的 $p(w|d)$ 的计算。对 Doc 中所有的 doc 中的所有 w 进行一次 $p(w|d)$ 的计算并重新选择 topic 看作一次迭代。这样进行 n 次循环迭代之后，就会收敛到

LDA 所需要的结果了。

M2: 程序实现

(一) 语料预处理。所用数据库为金庸小说，读取文本后需要进行一些处理包括：

- 1) 删除开头无用信息；
- 2) 删除中文停词及标点符号（利用 cn_stopwords.txt 文件）；
- 3) 删除空白符号，比如空格、换行符等。

程序如下：

```
def read_file(filename):
    # 如果未指定名称，则默认为类名
    target = "data/" + filename + ".txt"
    with open(target, "r", encoding='gbk', errors='ignore') as f:
        data = f.read()
        data = data.replace(
            '本书来自www.cr173.com免费txt小说下载站\n更多更新免费电子书请关注www.cr173.com', '')
        f.close()
    # 分词
    with open("tools/cn_stopwords.txt", "r", encoding='utf-8') as fp:
        stop_word = fp.read().split('\n')
        fp.close()
    split_word = []
    for words in jieba.cut(data):
        if (words not in stop_word) and (not words.isspace()):
            split_word.append(words)
    return split_word
```





(二) 段落抽取。题目要求从语料库中均匀抽取 200 个段落，每个段落大于 500 词，利用 jiebe 分词工具进行分词后，根据每篇小说的词数占总语料库词数的比例确定该篇小说应抽取的段落数，每篇小说抽取的段落数如下：

```
白马啸西风:2
碧血剑:11
飞狐外传:10
连城诀:5
鹿鼎记:28
三十三剑客图:2
射雕英雄传:21
神雕侠侣:23
书剑恩仇录:12
天龙八部:28
侠客行:8
笑傲江湖:22
雪山飞狐:3
倚天屠龙记:23
鸳鸯刀:1
越女剑:1
200
```

代码如下：

```
def extract_paragraph(is_word):
    # 读取小说名字
    word_len = 0
    with open("data/inf.txt", "r") as f:
        txt_list = f.read().split(',')
        dict = {}
        for name in txt_list:
            data = read_file(name, is_word)
            word_len += len(data)
            dict[name] = data
        f.close()
    # 计算每篇文章抽取段落数
    number = 1
    con_list = []
    for name in txt_list:
        count = int(len(dict[name]) / word_len * 200 + 0.5)
        # 特殊处理
        if name == '越女剑':
            count = 1
        pos = int(len(dict[name]) // count)
        for i in range(count):
            data_temp = dict[name][i * pos:i * pos + 500]
            con = {
                'number': number,
                'label': name,
                'data': data_temp
            }
            con_list.append(con)
            number += 1
```

分别以词和字为基本单元进行处理，将训练集和测试集分别保存为如下文件：

-  word.csv
-  word_tr.csv
-  words.csv
-  words_tr.csv

（三）初始化。将从每篇小说中抽取的所有段落组合到一起作为训练集，进行训练。首先为每篇文章中的词随机分配一个 topic，然后统计每篇文章的 topic 频率以及每个 topic 的词频。

```

def __init__(self, data_txt):
    self.topic_count = 16
    self.topic_word_count = {} # 每个topic有多少词
    self.topic_word_fre = {} # 每个topic的词频表，字典的列表
    for i in range(self.topic_count):
        self.topic_word_fre[i] = self.topic_word_fre.get(i, {})
    self.doc_word_from_topic = [] # 每篇文章中的每个词来自哪个topic
    self.doc_word_count = [] # 每篇文章中有多少词
    self.doc_topic_fre = [] # 每篇文章topic词频
    for data in data_txt:
        topic = []
        docfre = {}
        for word in data:
            a = random.randint(0, self.topic_count - 1) # 为每个单词赋予一个随机初始topic
            topic.append(a)
            if '\u4e00' <= word <= '\u9fa5':
                self.topic_word_count[a] = self.topic_word_count.get(a, 0) + 1 # 统计每个topic总词数
                docfre[a] = docfre.get(a, 0) + 1 # 统计每篇文章对应topic的词频
                self.topic_word_fre[a][word] = self.topic_word_fre[a].get(word, 0) + 1
        self.doc_word_from_topic.append(topic)
        for i in range(self.topic_count):
            docfre[i] = docfre.get(i, 0)
        docfre = list(dict(sorted(docfre.items(), key=lambda x: x[0], reverse=False)).values())
        self.doc_topic_fre.append(docfre)
        self.doc_word_count.append(sum(docfre)) # 统计每篇文章的总词数
    self.topic_word_count = list(
        dict(sorted(self.topic_word_count.items(), key=lambda x: x[0], reverse=False)).values())
    self.doc_topic_fre = np.array(self.doc_topic_fre) # 转为array方便后续计算
    self.topic_word_count = np.array(self.topic_word_count) # 转为array方便后续计算
    self.doc_word_count = np.array(self.doc_word_count) # 转为array方便后续计算
    self.doc_topic_pro = [] # 每个topic被选中的概率
    self.doc_topic_pro_new = [] # 记录每次迭代后每个topic被选中的概率
    self.cal_pro()

```

（四）训练。根据式(1.1)计算每个 topic 下的词出现在这个位置的概率，然后更改该词的 topic 为概率最大的 topic，随后进行文章 topic 频率以及 topic 词频的更新，当频率无变化时，说明算法已经收敛，迭代终止。

```

for data in data_txt:
    top = self.doc_word_from_topic[i]
    for w in range(len(data)):
        word = data[w]
        pro = []
        topfre = []
        if '\u4e00' <= word <= '\u9fa5':
            for j in range(self.topic_count):
                topfre.append(self.topic_word_fre[j].get(word, 0))
            pro = self.doc_topic_pro[
                i] * topfre / self.topic_word_count # 计算每篇文章选中各个topic的概率乘以该词语在每个topic中出现的概率，得到该词出现的概率向量
            m = np.argmax(pro) # 认为该词是由上述概率之积最大的那个topic产生的
            self.doc_topic_fre[i][top[w]] -= 1 # 更新每个文档有多少各个topic的词
            self.doc_topic_fre[i][m] += 1
            self.topic_word_count[top[w]] -= 1 # 更新每个topic的总词数
            self.topic_word_count[m] += 1
            self.topic_word_fre[top[w]][word] = self.topic_word_fre[top[w]].get(word, 0) - 1 # 统计每个topic总词数
            self.topic_word_fre[m][word] = self.topic_word_fre[m].get(word, 0) + 1 # 统计每个topic总词数
            top[w] = m
    self.doc_word_from_topic[i] = top
    i += 1

```

（五）测试。将 200 个段落利用训练好的 topic 词的分布计算段落的主题分布，并且利用欧氏距离选择该段落与哪一个样例更加接近，便认为该段落属于这一类。

```
result = []
for k in range(len(test_txt)):
    pro = []
    for i in range(len(data_txt)):
        dis = 0
        for j in range(self.topic_count):
            dis += (self.doc_topic_pro[i][j] - doc_topic_pro[k][j]) ** 2 # 计算欧式距离
        pro.append(dis)
    m = pro.index(min(pro))
    result.append(m)
print(result)
return result
```

Experimental Studies

选择主题数量为 16，200 个段落从新组合成 16 篇文档，每篇文档主题分布如下：

	± 0	± 1	± 2	± 3	± 4	± 5	± 6	± 7	± 8	± 9	± 10	± 11	± 12	± 13	± 14	± 15
0	0.04700	0.06800	0.09700	0.02200	0.06200	0.10800	0.01900	0.03100	0.04900	0.06100	0.12600	0.06100	0.12500	0.06200	0.02800	0.03400
1	0.04300	0.06741	0.08927	0.07069	0.06686	0.04445	0.03407	0.08854	0.11040	0.06486	0.06686	0.08508	0.05356	0.03607	0.03516	0.04372
2	0.04480	0.06740	0.06500	0.04600	0.04800	0.11760	0.04840	0.06500	0.05280	0.07320	0.09840	0.05400	0.05620	0.04800	0.05340	0.06180
3	0.07760	0.04840	0.06360	0.04640	0.07480	0.06280	0.07920	0.05840	0.06160	0.07320	0.06360	0.04280	0.07040	0.07560	0.04480	0.05680
4	0.06418	0.05260	0.07533	0.07111	0.05746	0.04788	0.06825	0.05825	0.05682	0.06261	0.05803	0.07990	0.05825	0.06404	0.06832	0.05696
5	0.05305	0.04705	0.05806	0.07908	0.08408	0.05005	0.08008	0.09409	0.05606	0.05305	0.06206	0.04605	0.09209	0.04805	0.03504	0.06206
6	0.05321	0.04786	0.07032	0.04930	0.06449	0.05274	0.07251	0.05474	0.07261	0.05809	0.06105	0.04529	0.06411	0.09640	0.06516	0.07213
7	0.06218	0.05453	0.07631	0.07109	0.07100	0.06182	0.06245	0.04994	0.04967	0.05543	0.08072	0.05075	0.07982	0.06713	0.04616	0.06101
8	0.08585	0.06668	0.07235	0.05484	0.04867	0.07168	0.06268	0.06894	0.06868	0.05534	0.06651	0.06218	0.05701	0.05301	0.05868	0.04701
9	0.06065	0.06594	0.06680	0.05815	0.06844	0.05172	0.07794	0.06322	0.06144	0.05422	0.07372	0.05529	0.05737	0.05136	0.07837	0.05537
10	0.06278	0.05628	0.05703	0.05528	0.05828	0.06428	0.04877	0.08704	0.05003	0.06228	0.06778	0.06653	0.07279	0.05303	0.05528	0.08254
11	0.07282	0.05736	0.06673	0.05455	0.07318	0.05900	0.04709	0.05418	0.05982	0.07855	0.06418	0.05945	0.06873	0.05409	0.06527	0.06500
12	0.05133	0.06000	0.06000	0.06200	0.05533	0.08067	0.06200	0.05533	0.06533	0.06133	0.07267	0.05467	0.05267	0.04800	0.07867	0.08000
13	0.05609	0.07412	0.06663	0.05531	0.07961	0.07003	0.05844	0.06175	0.05792	0.05540	0.05853	0.06010	0.05818	0.06184	0.06803	0.05801
14	0.05411	0.06814	0.05210	0.05812	0.06814	0.05210	0.04609	0.07816	0.05812	0.07615	0.06814	0.07615	0.09619	0.05210	0.04609	0.05010
15	0.13400	0.03800	0.06000	0.08200	0.06600	0.05400	0.06800	0.05400	0.05400	0.05200	0.07000	0.05800	0.10000	0.04400	0.03400	0.03200

计算 200 个段落的主题分布，随后进行分类，分类结果如下：

number:185	label:倚天屠龙记	result:倚天屠龙记
number:186	label:倚天屠龙记	result:倚天屠龙记
number:187	label:倚天屠龙记	result:三十三剑客图
number:188	label:倚天屠龙记	result:倚天屠龙记
number:189	label:倚天屠龙记	result:倚天屠龙记
number:190	label:倚天屠龙记	result:倚天屠龙记
number:191	label:倚天屠龙记	result:倚天屠龙记
number:192	label:倚天屠龙记	result:天龙八部
number:193	label:倚天屠龙记	result:倚天屠龙记
number:194	label:倚天屠龙记	result:倚天屠龙记
number:195	label:倚天屠龙记	result:倚天屠龙记
number:196	label:倚天屠龙记	result:倚天屠龙记
number:197	label:倚天屠龙记	result:射雕英雄传
number:198	label:倚天屠龙记	result:倚天屠龙记
number:199	label:鸳鸯刀	result:鸳鸯刀
number:200	label:越女剑	result:越女剑
分类正确数: 155 分类错误数: 45		

改变主题个数验证主题个数对结果的影响，由于初始化是随机初始化，每次运行结果不同，针对每个主题个数运行多次，结果如下：

Table 1: 实验结果（以词为基本单元）

主题个数	正确分类个数	错误分类个数	正确率	平均正确率
16	155	45	77.5%	79%
	153	47	76.5%	
	166	34	83%	
50	186	14	93%	92.6%
	184	16	92%	
	186	14	93%	
100	191	9	95.5%	96.7%
	193	7	96.5%	
	196	4	98%	

从表中可以看出，随着主题个数的增多，分类结果正确率不断增加。

下面是以字为基本单元进行训练测试的结果：

Table 2: 实验结果（以字为基本单元）

主题个数	正确分类个数	错误分类个数	正确率	平均正确率
16	106	96	52.48%	50.5%
	92	110	45.54%	
	108	94	53.47%	
50	145	57	71.78%	73.93%
	154	48	76.24%	
	149	53	73.76%	
100	171	31	84.65%	85.97%
	178	24	88.12%	
	172	30	85.15%	

Conclusions

根据表 1 可以看出，随着主题数量的增多，分类性能逐渐变好，相当于每篇文档比较的特征数目变多，比较效果更好。

根据表 2 可以看出，在相同主题数量的情况下，以字为基本单元的分类效果比以词为基本单元的分类效果差，这是因为针对不同主题，以字为基本单元的特异性较小，不同字之间的差异较小，而不同词的差异性较大，所以以词为基本单元分类效果更好。

References

本文 LDA 初始化、训练及测试代码参考该博客内容：https://blog.csdn.net/weixin_44966965/article/details/124556948。

与原代码相比，本文做了如下改进：

- 1) 采用类的结构进行重构，将训练、测试过程分开。
- 2) 修改原文代码 Bug，原代码初始化 topic 统计词频时统计的是固定 topic 的词频，即默认第一篇小说的词全部来自 topic_0。如图所示：

```
if '\u4e00' <= word <= '\u9fa5':  
    Topic_count[a] = Topic_count.get(a, 0) + 1 # 统计每个topic总词数  
    docfre[a] = docfre.get(a, 0) + 1 # 统计每篇文章的词频  
    exec('Topic_fre{word}=Topic_fre{}.get(word, 0) + 1'.format(i, i)) # 统计每个topic的词频
```

- 3) 本文实现了主题数量可调，原代码主题数量恒等于语料库小说数量，存在一定问题。