学　　期 2023-2024 第二学期

北京航空航天大學
BEIHANG UNIVERSITY

Deep NLP

第二次大作业

院（系）名称	自动化科学与电气工程学院
专业名称	控制工程
学生姓名	乔彪
学号	ZY2303706

2024 年 5 月

一、问题描述

从链接给定的语料库中均匀抽取 1000 个段落作为数据集（每个段落可以有 K 个 token, K 可以取 20，100，500, 1000, 3000），每个段落的标签就是对应段落所属的小说。利用 LDA 模型在给定的语料库上进行文本建模，主题数量为 T，并把每个段落表示为主题分布后进行分类（分类器自由选择），分类结果使用 10 次交叉验证（i.e. 900 做训练，剩余 100 做测试循环十次）。实现和讨论如下的方面：（1）在设定不同的主题个数 T 的情况下，分类性能是否有变化？；（2）以"词"和以"字"为基本单元下分类结果有什么差异？（3）不同的取值的 K 的短文本和长文本，主题模型性能上是否有差异？

二、问题解释

LDA（Latent Dirichlet Allocation）模型是一种基于概率图模型的文本主题分析方法。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。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：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四个步骤：

1) 从狄利克雷分布α中取样生成文档i主题分布θ_i；

2) 从主题的多项式分布θ_i中取样生成文档i 第j个词的主题$Z_{i,j}$；

3) 从迪利克雷分布β中取样生成主题$Z_{i,j}$对应的词语分布$\Phi_{Z_{i,j}}$；

4) 从词语的多项式分布$\Phi_{Z_{i,j}}$中采样最终生成词语$W_{i,j}$。

通过对这个过程进行反推，可以得到 LDA 模型的参数估计方法。具体来说，我们需要通过文本数据中观察到的单词来估计每个主题的单词分布以及每个文档的主题分布，然后通过这些参数来推断文本的标签。

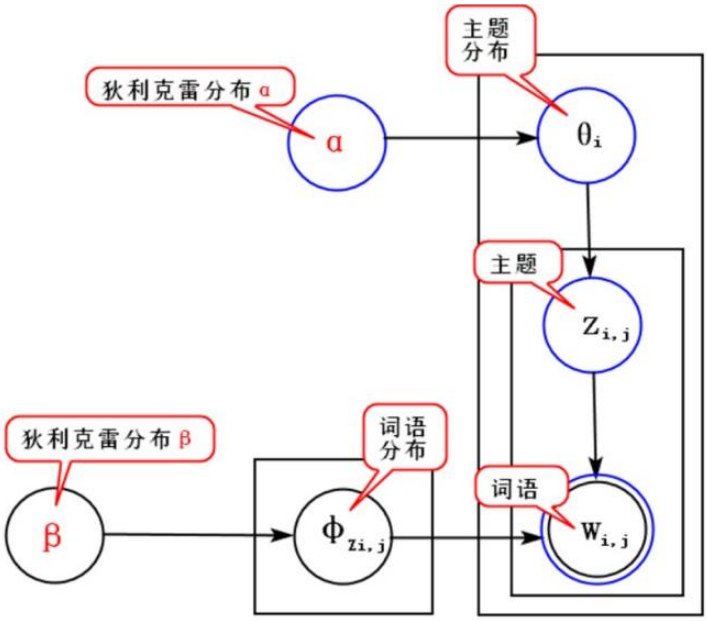

图 1 LDA 模型

三、代码实现

由于数据量处理量较多，本次大作业将实验代码分为了三个 python 文件。

（一）读取所有小说并保存

```
if __name__ == '__main__':

    punctuations = get_useless("cn_punctuation.txt")
    stopwords = get_useless("cn_stopwords.txt")

    texts ={}
    rootDir = 'jyxstxtqj_downcc.com'

    get_texts(texts,rootDir,punctuations,stopwords)
    with open('texts.json', 'w') as json_file:
        json.dump(texts, json_file)
```

这段代码的作用是读取语料库中所有小说，然后进行预处理，包括去除所有的如停顿词等的无用的字符，最后以字典的形式保存文本，字典的“key”为，每个小说名称，“value”为每个小说预处理后的文本。内容与上一次作业相似。

（二）建立 LDA 模型，验证 SVM 分类器效果

```python
if __name__ == '__main__':

    with open('texts.json', 'r') as json_file:
        texts = json.load(json_file)
    ks = [20,100,500,1000,3000]
    ts = [8,16,24,32]
    modes = ['以“词”为基本单元','以“字”为基本单元']
    rates = []
    for k in ks:
        for mode in modes:
            bags,labels = get_bags_and_labels(texts, k, mode, size: 1000)
            dictionary = corpora.Dictionary(bags)
            corpus = [dictionary.doc2bow(bag) for bag in bags]
            for t in ts:
                ldamodel = gensim.models.LdaMulticore(corpus, num_topics=t, id2word=dictionary, passes=10, workers=2)
                distributions = get_distribution(ldamodel, bags)
                rate = evaluate_classification(distributions, labels)
                rates.append((rate,k,t,mode))
    with open('rates.json', 'w') as json_file:
        json.dump(rates, json_file)
```

这段代码在读取上一段代码的文本字典后，依据不同的 token 值以及处理文本模型，先获得合适的段落以及对应标签；然后依据主题数量，利用 LDA 模型，获得每个词袋的主题分布；最后利用 SVM 分类器分类并保存分类效果，保存的分类准确率以四元组列表的形式保存，四元组中第一个元素是准确率，第二个元素是分段 token 数，第三个元素是主题数，第四个元素是选择文本处理模式，即以“词”为基本单元或是以“字”为基本单元。

1. get_bags_and_labels()

```python
def get_bags_and_labels(texts,k,mode,size):
    bags = []
    labels = []
    bags_dic = {}
    for name,text in texts.items():
        if mode == '以“词”为基本单元':
            text = jieba.lcut(text)
        text_l = len(text)
        for i in range(0,text_l, k):
            if i + k < text_l:
                bag = text[i:i + k]
                if mode == '以“字”为基本单元':
                    bag = list(bag)
                if name in bags_dic:
                    bags_dic[name].append(bag)
                else:
                    bags_dic[name] = [bag]
    bags_dic_c = {}
    for name,bags_seperated in bags_dic.items():
        if len(bags_seperated) >= 10:
            bags_dic_c[name] = bags_seperated
    bags_dic.clear()
    bags_l = len(bags_dic_c)
    size_seperateds = {key:(size//bags_l) for key in bags_dic_c.keys()}
    while sum(size_seperateds.values()) < size:
        random_key = random.choice(list(size_seperateds.keys()))
        size_seperateds[random_key] += 1
    compensate = 0
    for name, bags_seperated, size_seperated in zip(bags_dic_c.keys(), bags_dic_c.values(), size_seperateds.values()):
```

```
compensate = 0
for name, bags_seperated, size_seperated in zip(bags_dic_c.keys(), bags_dic_c.values(), size_seperateds.values()):
    bags_seperated_l = len(bags_seperated)
    if bags_seperated_l < size_seperated :
        compensate += (size_seperated - bags_seperated_l)
        size_seperateds[name] = bags_seperated_l
sorted_dict = dict(sorted(bags_dic_c.items(), key=lambda item: len(item[1]),reverse=True))
for name,bags_seperated in sorted_dict.items():
    if compensate == 0 :
        break
    if len(bags_seperated) >= compensate + size_seperateds[name]:
        size_seperateds[name] += compensate
        compensate = 0
    elif len(bags_seperated) > size_seperateds[name] and len(bags_seperated) < compensate + size_seperateds[name]:
        compensate -= (len(bags_seperated)-size_seperateds[name])
        size_seperateds[name] = len(bags_seperated)
for name,bags_seperated,size_seperated in zip(bags_dic_c.keys(),bags_dic_c.values(),size_seperateds.values()):
    bags_seperated_l = len(bags_seperated)
    step = bags_seperated_l // size_seperated
    if step == 0 :
        step = 1
    n = 0
    for i in range(0, bags_seperated_l,step):
        n += 1
        bags.append(bags_seperated[i])
        labels.append(name)
        if n >= size_seperated:
            break
return bags,labels
```

该函数可以分为两个部分，第一部分是获得每个小说各自的段落，由于交叉验证需要每个标签至少有 10 个，对于段落数不足 10 个的小说需要进行剔除。第二部分是尝试均匀的选取 1000 个段落。由于每个小说文本量差距过大，简单的采取随机取样的方法容易导致文本量小的小说的段落数不足 10 个。因此采取了较为复杂的办法来尽可能均匀的选取段落：首先给每个小说尽可能均匀的分配段落数；然后判断每个小说的段落数是否可以满足分配段落数的要求，若不满足要求，则优先由段落数较高的小说进行补偿段落数的差值；最后依据对每个小说分配的段落数，对每个小说分段采样，采样结果以段落列表和标签列表保存。

2. get_distribution()

```
def get_distribution(ldamodel, bags):
    distributions = []
    for bag in bags:
        bow_vector = ldamodel.id2word.doc2bow(bag)
        distribution = ldamodel.get_document_topics(bow_vector, minimum_probability=0.0)
        array = np.array([second for first, second in distribution])
        distributions.append(array)
    return distributions
```

该函数的作用是依据段落列表，利用 LDA 模型获得对应的主题分布列表。

3. evaluate_classification

```
def evaluate_classification(distributions, labels):
    cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
    scores = cross_val_score(SVC(), distributions, labels, cv=cv)
    return scores.mean()
```

该函数的作用是依据主题分布列表和标签列表，选择 SVM 分类器进行分类，分类结果使用 10 次交叉验证。最后返回准确率。

（三）绘制表格

```
if __name__ == '__main__':
    with open('rates.json', 'r') as json_file:
        rates = json.load(json_file)
    rates_dic ={}
    ks = [20,100,500,1000,3000]
    ts = [8,16,24,32]
    modes = ['以“词”为基本单元','以“字”为基本单元']
    paths = ['以“词”为基本单元.xlsx','以“字”为基本单元.xlsx']
    for rate in rates:
        rates_dic[(rate[1],rate[2],rate[3])] = rate[0]
    save_table(ks,ts,modes,paths,rates_dic)
```

该代码的作用是依据最后的准确率列表来绘制表格。首先读取上一段代码保存的四元组列表；然后将其转化为以三元组为“key”的字典，三元组中第一个元素是分段 token 数，第二个元素是主题数，第三个元素是选择文本处理模式，即以“词”为基本单元或是以“字”为基本单元。最后绘制表格。

```
def save_table(ks,ts,modes,paths,rates_dic):
    tables = {}
    ks_l = [f'k={num}' for num in ks]
    ts_l = [f't={num}' for num in ts]
    for mode in modes:
        table = []
        for k in ks:
            row = []
            for t in ts:
                row.append(rates_dic[(k,t,mode)])
            table.append(row)
        tables[mode] = table
    for mode,path in zip(modes,paths):
        df = pd.DataFrame(tables[mode], columns=ts_l)
        df.index = ks_l
        df.to_excel(path)
```

具体绘制列表的方法是将转化后的字典转化为以两个矩阵为元素的字典，字典的“key”为处理文本的模式。然后设置好保存路径、行标题和列标题后，输出两个列表。

四、运行结果

	t=8	t=16	t=24	t=32
k=20	0.077	0.077	0.067	0.048
k=100	0.106	0.079	0.13	0.083
k=500	0.225	0.233	0.272	0.239
k=1000	0.382	0.33	0.432	0.479
k=3000	0.703576	0.836057	0.76766	0.773072

表 1 以“词”为基本单元的分类器准确率

	t=8	t=16	t=24	t=32
k=20	0.093	0.087	0.085	0.078
k=100	0.199	0.198	0.23	0.222
k=500	0.408	0.54	0.569	0.661
k=1000	0.548	0.68	0.744	0.815
k=3000	0.759	0.85	0.921	0.94

表 2 以“字”为基本单元的分类器准确率

五、结果分析

当 token 数较小或以“词”为基本单元时，由于段落限制，随着主题数量增大，分类器准确率变化规律不太明显。当 token 数较大时，且以“字”为基本单元时，随着主题数量增大，准确率会随之增大。这说明主题数量增加一定程度上可以提高准确率。当以“词”为基本单元时，准确率的变化规律难以总结。k=3000，以“词”为基本单元时，准确率变化有先增大后减小的趋势，这说明如果主题数量过大可能导致过拟合，准确率反而下降。

随着 token 数增大时，准确率也会随之升高，说明随着段落长度的增大，信息量越多，越有助于分类。不过段落长度过大时，也会导致更多文本小的小说被剔除，这也可能增加准确率。

理论上“词”的语义信息要比“字”要多，区分度更高，但结果上以“词”为基本单元的准确率总是低于以“字”为基本单元的准确率。推测是因为这些文本出于同一个作者。