数据挖掘课程实验报告

Homework 1: VSM and KNN

201834879　王士东　2018/11/2

一. 实验主要工作介绍

1.预处理

首先将原数据集划分为训练数据集和测试数据集，其中训练数据占 80%，测试数据占 20%。对于每一个类别得多个多个对象，即一个文件夹的多个文档，先读取八个对象对放到训练数据集，再读取两个对象放到测试数据集，这样保证了测试数据集的均匀分布。（也可以设定随机函数，随机指定百分之八十的数据划分到训练数据集）

其中：

1）预处理函数 pretreat（）完成对数据集的划分，并调用写数据函数 writedata(path1,path2)将划分的数据写到对应的数据集；

2）写数据函数 writedata(path1,path2)：其中 path1 为原数据存储路径，path2 为处理后的数据的输出路径。然后用 nltk 进行一些文本的简单处理，如:去停词,提取词干,去掉数字只保留字母,并把所有大写字母转换为小写字母等。（装 nltk 文件包的时候，选择 ALL 一直报错，所以没有全选，只装了用到的包）

2.VSM

主要工作包括建立字典，计算 tf-idf 值和生成向量 Vector 等。

1）函数 creatidf()

主要功能是生成 value 值为 idf 的全局字典，其中为减少数据处理，通过计算词频（此处词频指词出现的次数）筛选掉一些词频较小得单词；

2）创建向量函数 creatvector(inpath,outpath)：其中 inpath 为原数据存储路径，outpath 为生成的向量的存储路径。

首先遍历文本，对每一个文本中的每一个单词计算 tf 的值，然后计算每一个单词 tf-idf 值，生成每一个文本的 tf-idf 字典，对 tf-idf 字典进行降序排列，此处只选取前 50 个关键字做为此文本的关键词，取他们的 tfidf 值，生成

新的 tfidf 字典，然后加上其所属的类别作为一个向量，此向量为将其作为一条数据加入到 list 集中，最后用 json 的方式，将 list 向量集存储到对应的 json 文件。

3.KNN

主要工作是设定 K 的值，算出每一对向量的相似程度（余弦大小 = AB / |A|*|B|），得出预测类别，比较得出预测准确率。

1）qiumo(vc)：vc 为从 json 文件中读出的 list，格式为：[[str,dict],[str,dict],[str,dict]]

主要功能为：对于每个向量求出其模的大小，存为 list

2）knn()：

首先设定 K 的值，然后对于测试集的每一个向量，让其与训练集里的所有向量进行比较，存为 list，list 中为许多个二元组（类名和 cos 值），取前 K 个与其最相似（cos 值大）的向量，统计出前 k 个中出现的最多次数的类别，此类别为模型的预测类别，让其与该向量的真实类别比较，若相同，即为预测成功，最后对于全部的测试向量，计算出预测成功次数除以总测试次数，即为预测成功率。

二．实验实现过程与问题分析

1.实现过程

首先是 python 语言的学习，由于之前没有系统的学过 python，于是根据课件上给的视频资料以及廖雪峰的 python 学习课程，系统的学习了下 python 的基础知识；

然后是对 VSM 和 KNN 的理解学习，又重新梳理了一遍老师课堂课件，并且读了一些网络上的相关博客，对 VSM 和 KNN 的原理有了详细的了解。

最后是模型的编程实现，第一是预处理工作，这部分主要做了划分数据集和 nltk 文本处理的工作，在此处用了 nltk 文本处理工具；第二是 VSM 模型向量的生成，这部分主要工作是 tf-idf 值的计算和向量词典的生成；第三是 KNN 算法的实现，这部分主要是设置 K 的值和将测试集向量与训练集进行比较，最后计算模型准确率。

2.问题分析

1）数据集预处理时用的是 nltk 工具，但是装文件包的时候一直报错，为了节省时间，所以只安装了本次实验所用的文件包，等实验结束再去检查安装过程哪里出了问题。

2）一开始读文件数据的时候，由于可能包含特殊字符，所以最好不要指定编码类型，或者指定一个较大的编码类型，但是我测试过还是有错，所以最后选择了忽略，还有文件夹写的时候，要有文件夹检测语句，看是否创建

3）词频是一个单词在文档中的出现次数，但是还要进行标准化，tf 值要用词频再除以文档中出现最多的那个单词的出现次数。还有，idf 是对整个数据集而言的（算他的时候，要给 df+1），tf 是对每一个文本文档而言得，搞清楚这个用了好长时间。

4）按行读出数据来，并不是直接就是单词，一定要去换行符和首尾空格，此处如果不去换行符，而 idf 字典里没有换行符，会影响下条判断语句执行。for s in lines: s = s.strip('\n')。

5）sorted()排序函数只对含有一组键值对的列表起作用，最后返回值类型也是 list，所以在处理完一定要记着手动把 list 转换为 dict，一开始忽略了这个问题，后来发现从 json 文件中读出来的数据不是 dict，才开始找问题，耗费了不少时间。还有就是没有 list 转 dict 的函数，只能自己写。

6)存向量的时候采用的 json 的存储方式，直接好处是可以直接存储数据类型,比如一个 list,一个 dict,并且最方便的是可以直接读出来,还是一个 list，一个 dict。本实验中，我的 json 中存储的格式为：[[str,dict],[str,dict],[str,dict]]，一个双层 list。

三. 模型性能测试与分析

1.性能分析

主要为三个参数：k 是 KNN 中设定的 k 值，vector 是生成的向量中保存的单词的个数, tf 是筛选数据的阈值。

1）为了减少数据处理的规模，所以对数据进行了筛选，可以筛选掉出现次数小于某个值的单词，但是经测试，相应的检测准确率会随之减小，还有，这个

数值不能过大，比如这个值为10，而一个文档中一共20个单词，并且这个关键词是这个文档的关键词，然后正好在这个文件出现了8次，然后要是按照设定的规则就得把他删了。所以觉得，这个阈值最好为1或者2（潜意识默认出现一次两次不可能为关键词），这个时候数据规模为20000~40000之间，其实不筛选也行，准确率更高一点。

2）生成向量的时候，按照 tf-idf 值对字典进行了降序排列，只取了前 50 个作为关键字存储到了向量里，这样大大减少了在执行 knn 的时候数据的处理规模，并且经测试关键字过多，或者不做限制的话对准确率的提升甚微，但是，如果关键字选取过少，对结果有较大影响，数据集里单词大约在 30~500 之间，大部分不是很多，本实验选区的 50 个关键字，下面有具体的测试。

3）比较向量的相似性，是算的两个向量的 cos 值。先将其存到字典，然后在对其进行排序。有两种方式，第一种方式是先存下所有的值，然后进行排序，然后再选取前 k 个；第二种方式是只保存 k 个，每次都用新的来和之前 k 个中最小的那个进行比较，若是大就更新。显然，第二种的时间规模小于第一种，可能当数据规模很大的时候能够体现出差别，但是，测试完，发现第一种方法要比第二种高一个百分点。一开始我以为是内置 sorted（）函数在比较相等的值得时候也交换了对象，于是去更改了第二种方法，让其在相等时候也交换对象，发现准确率无变化。现在还不知道原因，接下来再考虑。

2.性能测试

不同参数条件下下的模型运行结果：

字典大小：82599　tf ＞ 0；字典大小：14911　tf ＞ 10；

字典大小: 47537　tf ＞ 1；字典大小: 21860 tf ＞ 5。

1）模型准确率最高：

k = 3　vector = 50　tf ＞ 1

总测试次数：　3754

预测成功次数：　3207

预测失败次数：　547

预测准确率：　0.854288758657432

2）模型准确率最低：

k = 10 vector = 30 tf > 10

总测试次数： 3754

预测成功次数： 3035

预测失败次数： 719

预测准确率： 0.8084709643047416

3）其他参数组合下模型准确率：

k = 5 vector = 50 tf > 5

总测试次数： 3754

预测成功次数： 3167

预测失败次数： 587

预测准确率： 0.8436334576451785

k = 5 vector = 50 tf > 1

总测试次数： 3754

预测成功次数： 3179

预测失败次数： 575

预测准确率： 0.8468300479488545

k = 10 vector = 200 tf > 0

总测试次数： 3754

预测成功次数： 3092

预测失败次数： 662

预测准确率： 0.823654768247203

k = 10 vector = 50 tf > 10

总测试次数： 3754

预测成功次数： 3102

预测失败次数： 652

预测准确率： 0.8263185935002664

k = 10 vector = 50 tf > 0

总测试次数： 3754

预测成功次数： 3076

预测失败次数： 678

预测准确率： 0.8193926478423016

k = 15 vector = 50 tf > 0

总测试次数： 3754

预测成功次数： 3071

预测失败次数： 683

预测准确率： 0.8180607352157698

k = 20 vector = 50 tf > 0

总测试次数： 3754

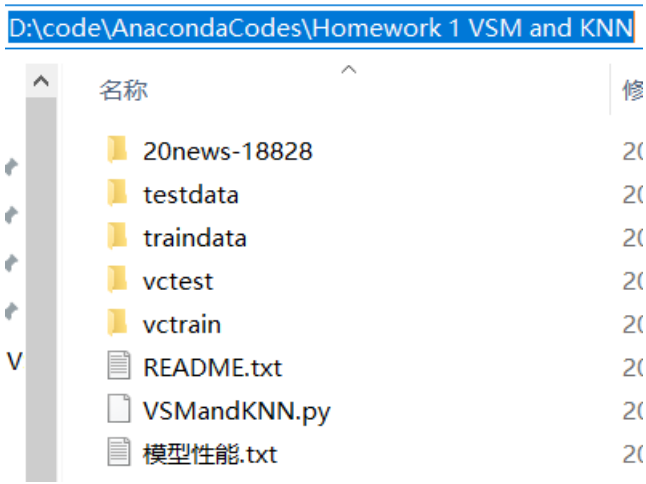
预测成功次数： 3056

预测失败次数： 698

预测准确率： 0.8140649973361748

四. 程序实现

1.程序相关文档



其中，testdata 和 traindata 分别为训练集和测试集，vctest 和 vctrain 分别存放训练集和测试集的模型向量，VSMandKNN.py 为全部程序代码。

2.json 文件中的数据格式


```
[
    ["alt.atheism", {
        "moral": 1.5308455408211512,
```

```
        "murder": 1.4834617287413534,
        "object": 0.9305039245129898,
        "system": 0.6331328029416594,
        "keith": 0.5714451739832879,
        "livesey": 0.5584364596964005,
        "goal": 0.5497706035527054,
        "arbitrari": 0.38681600241741704,
        "defin": 0.38149554248311196,
        "schneider": 0.36450040587387933,
        "golen": 0.32309987627264913,
        "kuweit": 0.32309987627264913,
        "arbitari": 0.3084256046846757,
        "jon": 0.29906885749255663,
        "violat": 0.29847311409950694,
        "exampl": 0.2913478215576497,
        "eactli": 0.2899382088833127,
        "claim": 0.28761782656130436,
        "pillar": 0.2777608725767928,
        "propog": 0.2614029854814621,
        "recurs": 0.2553570965524112,
        "prc": 0.25017810432334076,
        "whit": 0.2397665429393158,
        "definit": 0.23667898149473016,
    }],
    [str,{dict}],
    [str,{dict}]
    ...

]
```

3. 主要代码展示

```
#预处理函数
def pretreat():
    #读文件
    rootpathlist = os.listdir(oripath)
    for i in rootpathlist:
        rootpath = oripath + os.path.sep + i #目录名+路径切割符+文件名;i为每一个主文件夹的路径
        subspathlist = os.listdir(rootpath) #子文件夹列表
        k=1;
        for j in subspathlist:
            orisubspath = oripath + os.path.sep + i + os.path.sep + j
            if(k>=9):
                if os.path.exists(testdata + os.path.sep + i)==False:
                    os.mkdir(testdata + os.path.sep + i)
                    #如果不检测目录有无创建，会报错找不到目录
                testsubspath = testdata + os.path.sep + i + os.path.sep + j

                #调用写数据函数
                writedata(orisubspath,testsubspath)
            else:
                #print ('%s %s' % (i,j))
                if os.path.exists(traindata + os.path.sep + i)==False:
                    os.mkdir(traindata + os.path.sep + i)
                trainsubspath = traindata + os.path.sep + i + os.path.sep + j

                writedata(orisubspath,trainsubspath)
            k = k + 1
            if(k > 10):
                k = 1
#写数据函数
def writedata(path1,path2):
    #写数据
    openw = open(path2,'w')
    openr = open(path1,'r',errors = 'ignore')
    #指定编码类型或者默认都报编码错误，由于文件内有特殊字符，所以选择忽略
    datalines = openr.readlines()  #按行读，每一行作为一个处理单位
    openr.close()

    #用nltk进行去停词，提取词干，去掉数字等只保留字母,并一律按小写字母处理
    for line in datalines:
        stopwords = nltk.corpus.stopwords.words('english')
        porter = nltk.PorterStemmer()
        filterwords = re.compile('[^a-zA-Z]')
        wordsum = [porter.stem(word.lower()) for word in filterwords.split(line) \
                   if len(word)>0 and word.lower() not in stopwords]
        for s in wordsum: #对于这一行的每一个单词，调用一次写操作
            openw.write('%s\n' % s) #加入换行符，一个单词占一行
    openw.close()
```

```python
#生成value值为idf的字典
def creatidf():

    #计算词频（单词总的出现次数）及Idf（整个数据集中有多少个文本包含这个单词）
    #生成按照词频筛选后的字典
    worddict = {}  #每个单词总的出现次数
    dfdict = {}    #每个单词的df
    worddfdict = {} #筛选后每个单词的df
    wordidfdict = {} #value值为idf的字典

    numfiles = 0 #总文档个数

    rootpathlist = os.listdir(traindata)
    for i in rootpathlist:
        rootpath = traindata + os.path.sep + i #目录名+路径切割符+文件名;i为每一个主文件夹的路径
        subspathlist = os.listdir(rootpath) #子文件夹列表

        numfiles = numfiles + len(subspathlist)  #计算总文档个数

        for j in subspathlist:
            subspath = rootpath + os.path.sep + j
            lines = open(subspath).readlines() #此时每一行都为一个单词

            #计算单词出现次数
            for s in lines:
                #print(s)
                s = s.strip()
                if s in worddict:
                    worddict[s] = worddict[s] + 1
                else:
                    worddict[s] = 1
            #计算df
            coun = collections.Counter(lines) #counter函数
            for key,value in coun.items():  # counter函数的items()转化成(元素，计数值)组成的列表
                key = key.strip('\n')
                #print(key)
                #dfdict[key] = dfdict.get(key,0) + 1       #计算原始df
                if key in dfdict:
                    dfdict[key] = dfdict[key] + 1
                else:
                    dfdict[key] = 1

    #对字典做一个筛选，把那些出现次数较小的单词删去，然后赋值新的df
    print('筛选前字典大小:',len(dfdict))
    for key, value in worddict.items():
        if value > 0:
            # print(dfdict[key])
            worddfdict[key] = dfdict[key]
    #print(len(worddfdict))

    #maxvalue = worddict[max(worddict,key = worddict.get)]  #该文出现次数最多的词出现的次数

    #计算idf逆文档频率 = log(语料库的文档总数/(包含该词的文档数 + 1)),+1是为了防止分母为0，即所有文
    for key,value in worddfdict.items():
        #wordtfidfdict[key] = (worddict[key]/maxvalue) * (math.log(numfiles/(value + 1)))
        wordidfdict[key] = math.log10(numfiles/(value + 1))
    return wordidfdict
```

```python
#创建训练集和测试集的模型向量的函数
def creatvector(inpath,outpath):
    wordidfdict = creatidf() #获得value值是idf值得字典
    print('筛选后字典大小:',len(wordidfdict))
    rootlist = []  #rootlist格式为：[[str,dict],[str,dict],[str,dict]]

    rootpathlist = os.listdir(inpath)
    for i in rootpathlist:
        rootpath = inpath + os.path.sep + i #目录名+路径切割符+文件名;i为每一个主文件夹的路径
        subspathlist = os.listdir(rootpath) #子文件夹列表
        for j in subspathlist:
            subslist = []       #格式为：[str,dict]
            subslist.append(i)  #存放类名，i是类名，j是文本文档名
            wordtfidfdict = {}  #value值为tfidf的字典
            worddict = {}       #value值为此文本文件的每一个单词的tf值
            subspath = rootpath + os.path.sep + j
            lines = open(subspath).readlines() #此时每一行都为一个单词，lines为这个文本文件中全部单词

            #计算tf.采用词频标准化    词频 = 某个词在文章中出现的次数/该文出现次数最多的词出现的次数
            #计算TF-IDF = 词频（TF）* 逆文档频率（IDF）
            #print(lines)
            for s in lines:
                s = s.strip('\n')# 此处如果不去换行符，idf字典里没有换行符，那么worddict将会是空的，因为下条语句不执行
                if s in wordidfdict:
                    #print(s)
                    #s = s.strip()
                    if s in worddict:
                        worddict[s] = worddict[s] + 1
                    else:
                        worddict[s] = 1

            #print(len(worddict))
            index = max(worddict, key=worddict.get)
            maxvalue = worddict[index]  #该文出现次数最多的词出现的次数

            #生成value为tf-idf的字典
            for key in worddict:
                wordtfidfdict[key] = (worddict[key]/maxvalue) * (wordidfdict[key])
            #print(type(wordtfidfdict))
            #print(len(wordtfidfdict))

            #降序排列，每个向量里只包含前50个词,并且把得到的元组列表重置为字典形式，必须转为字典，不然读数据较麻烦
            valueslist = sorted(wordtfidfdict.items(),key = lambda item:item[1],reverse=True)
            valuesdict = {}
            m = 0
            for key,value in valueslist:
                if m >= 50:
                    break
                m = m + 1
                valuesdict[key] = value
            #print(valuesdict)
            #valuesdict = sorted(wordtfidfdict.items(),key = lambda item:item[1],reverse=True)
            #print(type(valuesdict))

            subslist.append(valuesdict)     #存放value值为tf-idf的字典 [str,dict]
            rootlist.append(subslist)       #[[str,dict],[str,dict],[str,dict]]

    #print(rootlist)
    #将list写入json文件
    openw = open(outpath,'w')
    json.dump(rootlist,openw,ensure_ascii=False)
    openw.close()
```

```
#对数据集求每一个向量的模，返回一个list
def qiumo(vc):
    vcmo = [] #数据集每一个向量的模的list

    #print(type(vc))  list  [[str,dict],[str,dict],[str,dict]]
    #print(type(vc[0]))  list [str,dict]
    #print(type(vc[0][0]))  str
    #print(type(vc[0][1]))  dict

    for i in vc:  #i为 list [str,dict]
        mosum = 0
        #print(type(i[1]))
        for key in i[1]:
            mosum = mosum + i[1][key] * i[1][key]
        mo = math.sqrt(mosum)
        vcmo.append(mo)
    return vcmo
#KNN的具体实现
def knn():

    k = 3

    #从json文件中读取训练和测试模型向量
    openr = open(vctrainpath,'r')
    vctrain = json.load(openr)
    #print(vctrain)

    openr.close()
    #print (vctrain)
    openr = open(vctestpath,'r')
    vctest = json.load(openr)
    openr.close()
    #print (vctest)

    #先把每一个向量的模求出来，存为两个列表
    motrain = qiumo(vctrain)
    motest = qiumo(vctest)

    #预测成功与失败的次数
    success = 0
    failure = 0

    for i in range(len(vctest)):

        cos = []

        for j in range(len(vctrain)):

            temp = []

            vcsum = 0 #求向量的乘积
            for key in vctest[i][1]:
                if key in vctrain[j][1]:
                    vcsum = vcsum + vctest[i][1][key] * vctrain[j][1][key]
            #求余弦 = AB/|A|*|B|
            cosij = vcsum /( motrain[j] * motest[i])
```

```
            #
            temp.append(vctrain[j][0])
            temp.append(cosij)
            cos.append(temp)

        #筛选出前k个数据，同时把前k个数据，由二元组转化为字典，关键字为类名
        #降序排列
        coslist = sorted(cos,key = lambda item:item[1],reverse=True)


        cosdict = {}
        m = 0
        for key,value in coslist:
            if m >= k:
                break
            m = m + 1
            cosdict[key] = cosdict.get(key,0) + 1

        #print(m)
        #算出k个数据中出现次数最多的类
        maxclass = ' '
        maxvalue = 0

        for key,value in cosdict.items():
            if value > maxvalue:
                maxclass = key
                maxvalue = value

        #与测试集这一条数据的类别进行对比
        if maxclass == vctest[i][0]:
            success = success + 1
            #print('预测成功')
        else:
            failure = failure + 1
            #print('预测失败')
    #预测的成功率
    successp = success / (success + failure)

    print('该模型的性能：')
    print('总测试次数：' , (success + failure))
    print('预测成功次数：' , success)
    print('预测失败次数：' , failure)
    print('预测准确率：' , successp)
```

数据挖掘课程实验报告

Homework 2: NBC

201834879　王士东　2018/11/14

一．贝叶斯定理

概率论中的经典条件概率公式：

$$P(Y|X)=\frac{P(X|Y)P(Y)}{P(X)}$$

其中，P（X，Y）= P（Y，X）<=> P（X|Y）P（Y）= P（Y|X）P（X），即 X 和 Y 同时发生的概率与 Y 和 X 同时发生的概率一样。

二．朴素贝叶斯定理

朴素贝叶斯的经典应用是对垃圾邮件的过滤，以及对文本格式的数据进行处理，因此这里以此为背景讲解朴素贝叶斯定理。

设 D 是训练样本和相关联的类的集合，其中训练样本的属性集为 X{ X1,X2, ... , Xn}，共有 n 个属性；类集为 C{ C1,C2, ... ,Cm }，有 m 种类别。

朴素贝叶斯定理：

$$P(C_i/\boldsymbol{X})=\frac{P(\boldsymbol{X}/C_i)P(C_i)}{P(\boldsymbol{X})}.$$

其中，P（Ci|X）为后验概率，P（Ci）为先验概率，P（X|Ci）为条件概率。由于对于所有的测试集计算时，上式的分母都是一样的，都是 P(X=X(test))，所以只需考虑分子的最大值。

又由于朴素贝叶斯的两个假设：1、属性之间相互独立；2、每个属性同等重要。通过假设 1 知，条件概率 P（X|Ci）可以简化为：

$$\mathrm{P}\left(\mathrm{X}|\mathrm{C_i}\right)=\prod_{k=1}^{k=n}P(\mathrm{X_k}|\mathrm{C_i})=P(X_1|C_i)\times P(X_2|C_i)\times\cdots\times P(X_n|C_i)$$

三．朴素贝叶斯算法实现

朴素贝叶斯算法的核心思想，是选择训练集中具有最高后验概率的类别作为

测试数据的预测类别。下面介绍利用 Python 语言实现朴素贝叶斯算法的过程，其本质是利用词和类别的联合概率来预测给定文档属于某个类别。

1. 数据集的准备

本实验所用数据集已经预处理完成，共包含 20 个类别，每个类别中包含若干文本文档，每个文本文档里包含若干单词，每个单词占一行。

1）生成训练集向量

此过程返回一个训练集向量集合，list 的格式为：[[str,int,float,dict],[],[]]。其中，str 为某个类别，int 为该类别所有文档的单词总数（一个单词在一个文档中只计算一次），float 为 P(Ci)的值（该类别的文档总数/数据集文档总数），dict 为该类别的字典，字典的 value 值为该类别所有单词的 df 值（包含该单词的文档总数）。

```
#返回训练集向量集合
def gettrainlist():

    #list格式为：[[str,int,float,{}],  [],[]],
    #其中，str为某个类别，int为该类别所有文档的单词总数（一个单词在一个文档中只计算一次），
    #float为P(Ci)的值（该类别的文档总数/数据集文档总数），
    #dict为该类别的字典，字典的value值为该类别所有单词的df值（包含该单词的文档总数）。
    rootlist = []
    numfiles = getnumfiles(traindata) #总文档个数

    rootpathlist = os.listdir(traindata)
    for i in rootpathlist:
        rootpath = traindata + os.path.sep + i #目录名+路径切割符+文件名;i为每一个主文件夹的路径
        subspathlist = os.listdir(rootpath) #子文件夹列表

        dfdict = {} #存放值为df的类别字典
        subslist = [] #每一个类别的向量list,格式为[str,int,float,{}]
        #存类名
        subslist.append(i)
        #存各文档的单词总数（每个单词在一个文档中只计算一次）
        wordsum = 0
        for j in subspathlist:
            subspath = rootpath + os.path.sep + j
            lines = open(subspath).readlines() #此时每一行都为一个单词
            coun = collections.Counter(lines) #counter函数
            wordsum = wordsum + len(coun)
            for key,value in coun.items():  # counter函数的items()转化成(元素，计数值)组成的列表
                key = key.strip('\n')   #去除换行符，此时的数据是带着换行符的
                dfdict[key] = dfdict.get(key,0) + 1
        subslist.append(wordsum)
        #存P(Xi=c)
        pc = len(subspathlist)/numfiles #P(Xi=c)为该类别文档个数/总文档个数
        subslist.append(pc)
        #存字典
        subslist.append(dfdict)
        rootlist.append(subslist)
    return rootlist
```

2）生成测试集向量

此过程返回一个测试集向量集合，list 的格式为：[[str,list], [],[]]。其中，str 为测试集该向量所属的类别，list 为测试集该向量所包含的所有单词集合（无重复单词）。

```
#返回测试集向量集合
def gettestlist():

    #list格式为[[str,[]], [],[]],
    #其中，str为测试集该向量所属的类别，list为测试集该向量所包含的所有单词集合（无重复单词）。
    rootlist = []

    rootpathlist = os.listdir(testdata)
    for i in rootpathlist:
        rootpath = traindata + os.path.sep + i
        subspathlist = os.listdir(rootpath)

        for j in subspathlist:
            subslist = []   #格式为[str,[]]
            #存类名
            subslist.append(i)
            templist = []  #第一次subslist和templist写到j循环的外边，所以运行了三小时。。。。

            subspath = rootpath + os.path.sep + j
            lines = open(subspath).readlines() #此时每一行都为一个单词
            coun = collections.Counter(lines) #counter函数
            for key,value in coun.items():
                key = key.strip('\n')
                templist.append(key)

            #存单词list
            subslist.append(templist)
            rootlist.append(subslist)
    return rootlist
```

2.NBC 算法实现

1）装载数据：导入第一步已经预处理完成的训练集和测试集的向量集合

2）模型应用（伯努利朴素贝叶斯模型）：本实验采用的伯努利朴素贝叶斯模型，统计（训练）时和判断（测试）时均不考虑重复单词；

3)Laplace 平滑：若测试数据中有的单词在训练集没有出现，其概率就是 0，会十分影响分类器的性能，所以采取 Laplace 平滑，让分子各单词的出现次数默认加 1，让分母单词出现总数加上测试数据的单词总数，这样处理后不影响相对大小。

4）解决下溢问题：若很小是数字相乘，则结果会更小，再四舍五入存在误差，而且会造成下溢出。所以对概率值取 log，乘法变为加法，并且相对大小趋

势不变。

5）模型性能评测：记录模型预测成功与失败次数，计算并输出模型预测准确率。

```
def NBC():
    #装载数据
    trainlist = gettrainlist() #[[str,int,float,{}],  [],[]]
    print('训练集装载完成！ 数据集大小： ',len(trainlist))
    testlist = gettestlist() #[[str,[]],  [],[]]
    print('测试集装载完成！ 数据集大小： ',len(testlist))

    success = 0 #记录模型预测成功的次数
    failure = 0 #记录模型预测失败的次数

    print('预测开始：')
    for i in range(len(testlist)):
        maxp = 0          #与训练集向量比较后，最大的概率值P
        maxclass = ' '    #最大概率值P所属类的类名

        for j in range(len(trainlist)):
            #对P做log处理，不影响大小关系，不然的话用乘积，超过了计算机下限，最后全是0
            p = math.log10(trainlist[j][2])
            #print(p)
            for key in testlist[i][1]:
                #Laplace平滑，分子加1，分母加单词总数
                tempdp = trainlist[j][3].get(key,0) + 1
                tempfenmu = trainlist[j][1] + len(testlist[i][1])
                tempp = math.log10(tempdp / tempfenmu)

                p = p + tempp

            #让maxp,和maxclass初始值默认为第一个测试数据的值，不可以直接设为0，因为maxp的值可能小于0
            if j == 0:
                maxp = p
                maxclass = trainlist[j][0]
            elif p > maxp:
                maxp = p
                maxclass = trainlist[j][0]


        #print(maxclass,' == ', testlist[i][0])
        if maxclass == testlist[i][0]:
            success = success + 1
            #print('预测成功')
        else:
            failure = failure + 1
            #print('预测失败')

        if (i % 1000) == 0:
            print('已完成测试次数： ' ,i+1 ) #打印程序运行程度

    #输出模型的性能
    successp = success / (success + failure)
    print('预测结束！')
    print('该模型的性能：')
    print('总测试次数： ' , (success + failure))
    print('预测成功次数： ' , success)
    print('预测失败次数： ' , failure)
    print('预测准确率： ' , successp)
```

3.模型预测结果

```
In [2]: runfile('D:/code/AnacondaCod
AnacondaCodes/Homework 2 NBC')
训练集装载完成！  数据集大小：   20
测试集装载完成！  数据集大小：   15074
预测开始：
已完成测试次数：  1
已完成测试次数：  1001
已完成测试次数：  2001
已完成测试次数：  3001
已完成测试次数：  4001
已完成测试次数：  5001
已完成测试次数：  6001
已完成测试次数：  7001
已完成测试次数：  8001
已完成测试次数：  9001
已完成测试次数：  10001
已完成测试次数：  11001
已完成测试次数：  12001
已完成测试次数：  13001
已完成测试次数：  14001
已完成测试次数：  15001
预测结束！
该模型的性能：
总测试次数：  15074
预测成功次数：  14168
预测失败次数：  906
预测准确率：  0.9398965105479634
```

四.总结

不同于其它分类器，朴素贝叶斯是一种基于概率理论的分类算法。特征之间的条件独立性假设，显然这种假设显得“粗鲁”而不符合实际，这也是名称中“朴素”的由来，然而事实证明，朴素贝叶斯在有些领域很有用，比如垃圾邮件过滤。

在具体的算法实施中，要考虑很多实际问题。比如因为“下溢”问题，需要对概率乘积取对数；再比如词集模型和词袋模型，还有停用词和无意义的高频词的剔除，以及大量的数据预处理问题等。

总的来说，朴素贝叶斯原理和实现都比较简单，学习和预测的效率都很高，是一种经典而常用的分类算法。

数据挖掘课程实验报告

Homework 3: Clustering

201834879　　王士东　　2018/12/19

一. 实验主要内容

测试 scikit-learn 中以下聚类算法在 Tweets 数据集上的聚类效果，使用 NMI(Normalized Mutual Information)作为评价指标。

Method name	Parameters	Scalability	Usecase	Geometry (metric used)
K-Means	number of clusters	Very large `n_samples`, medium `n_clusters` with MiniBatch code	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity propagation	damping, sample preference	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with `n_samples`	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Spectral clustering	number of clusters	Medium `n_samples`, small `n_clusters`	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints	Distances between points
Agglomerative clustering	number of clusters, linkage type, distance	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance
DBSCAN	neighborhood size	Very large `n_samples`, medium `n_clusters`	Non-flat geometry, uneven cluster sizes	Distances between nearest points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers

二. 实验实现过程

1.数据集的准备

本实验所用数据集为 Tweets 数据集（为了使数据读取过程更易，本实验将数据集处理为一行存取一个字典），预处理过程可细分为以下三个部分。首先从数据集读取数据，并放到指定集合（text 文本集和初始 cluster 集）；然后使用分词函数（包括 jieba 分词函数和 nltk 分词函数）对数据集进行处理，并计算 tfidf 值；最后将计算得到的 tfidf 数组，初始类标签集和类别总数进行输出。代码实现如下：

```
def pretreat():

    #Tweets数据集的读取
    openr = open('E:\\codes\\AnacondaCodes\\Homework 3\\Tweets.txt','r')
    #此处为方便读取数据集，把Tweets数据集中的每一个dict都调整为一行储存
    datalines = openr.readlines()
    openr.close()
    textlist = []
    clusterlist = []
    k = 0
    for line in datalines:
        tempdict = json.loads(line.strip())
        textlist.append(tempdict['text'])
        clusterlist.append(tempdict['cluster'])
        if tempdict['cluster'] > k:
            k = tempdict['cluster']
    #print(textlist)
    #print(k) #输出结果k=110，k为在初始类标签中，最大的类标号，喻指最多类别总数

    #数据集的预处理
    tfidf_vectorizer = TfidfVectorizer(tokenizer=word_tokenize, stop_words='english',lowercase=True)
    '''  tokenizer: 指定分词函数; lowercase: 在分词之前将所有的文本转换成小写,本实验的数据集已处理为小写

    tfidf_matrix = tfidf_vectorizer.fit_transform(textlist)  #需要进行聚类的文本集
    tfidf_array = tfidf_matrix.toarray() #将数据集由矩阵形式转为数组
    #print(type(tfidf_array))
    #print(tfidf_matrix)

    return tfidf_array, clusterlist, k
```

2. sklearn 中各个聚类算法的实现

首先调用预处理函数，获得数据集向量，初始类标签和聚类数；然后调用 sklearn 中各个聚类函数对 Tweets 数据集进行聚类处理；最后比较各个算法在数据集上的聚类效果，使用 NMI 作为聚类效果的评价标准。代码实现如下：

```
def clusterrun():
    #装载数据
    print('开始数据装载: ')
    datalist, clusterlist, k = pretreat()
    print('数据装载完成! ')
    print('开始执行聚类算法(使用NMI作为评价标准): ')

    #kmeans算法
    cluster = KMeans(n_clusters = k)
    # init='k-means++', 加上参数init，用k-means++方法，nmi降低了一个百分点
    # n_clusters: 指定K的值;  init: 制定初始值选择的算法
    #返回各自文本的所被分配到的类索引
    clusterresult = cluster.fit_predict(datalist)
    nmi = normalized_mutual_info_score(clusterresult, clusterlist)
    print ('kmeans算法:',nmi)
    #0.7749669551975551
```

```
#AffinityPropagation算法
cluster = AffinityPropagation()
clusterresult = cluster.fit_predict(datalist)
nmi = normalized_mutual_info_score(clusterresult, clusterlist)
print ('AffinityPropagation算法:',nmi)
#0.777159260731288

#MeanShift算法
cluster = MeanShift(bandwidth=0.5, bin_seeding=True)
clusterresult = cluster.fit_predict(datalist)
nmi = normalized_mutual_info_score(clusterresult, clusterlist)
print ('MeanShift算法:',nmi)
#0.73317315060083

#SpectralClustering算法
cluster = SpectralClustering(n_clusters=k,assign_labels="discretize",\
                             random_state=0)
clusterresult = cluster.fit_predict(datalist)
nmi = normalized_mutual_info_score(clusterresult, clusterlist)
print ('SpectralClustering算法:',nmi)
#0.7815326108789783

#Ward hierarchical clustering算法
cluster = AgglomerativeClustering(n_clusters=k, linkage='ward')
clusterresult = cluster.fit_predict(datalist)
nmi = normalized_mutual_info_score(clusterresult, clusterlist)
print ('Ward hierarchical clustering算法:', nmi)
#0.7811756130463107

#AgglomerativeClustering算法
cluster = AgglomerativeClustering(n_clusters=k,linkage = 'average')
clusterresult = cluster.fit_predict(datalist)
nmi = normalized_mutual_info_score(clusterresult, clusterlist)
print ('AgglomerativeClustering算法:', nmi)
#0.8229597609328941

#DBSCAN算法
cluster = DBSCAN(eps = 0.95, min_samples = 1 )
clusterresult = cluster.fit_predict(datalist)
nmi = normalized_mutual_info_score(clusterresult, clusterlist)
print ('DBSCAN算法:', nmi)
#0.7658091617297429

#GaussianMixture算法
cluster = GaussianMixture(n_components=k, covariance_type='diag')
#当type为full（默认）时，会报错，难道有的组件求不出一般协方差矩阵
clusterresult = cluster.fit(datalist).predict(datalist)
nmi = normalized_mutual_info_score(clusterresult, clusterlist)
print ('GaussianMixture算法:', nmi)
#0.7804543792671195
print('聚类算法执行结束！')
```

3.各个聚类算法的效果对比：

输出 jieba 分词的结果，发现其中含有大量的空格字符，于是又用 nltk 做了分词，试验结果表明使用 nltk 分词对于某些聚类算法会出现较好的结果。

1）使用 jieba 分词的聚类效果：

```
In [3]: runfile('E:/codes/AnacondaCodes/Homework 3/sklearn.py'
开始数据装载:
数据装载完成!
开始执行聚类算法(使用NMI作为评价标准):
kmeans算法: 0.7716101520056072
AffinityPropagation算法: 0.777159260731288
MeanShift算法: 0.73317315060083
SpectralClustering算法: 0.7815326108789783
Ward hierarchical clustering算法: 0.7811756130463107
AgglomerativeClustering算法: 0.8229597609328941
DBSCAN算法: 0.7658091617297429
GaussianMixture算法: 0.7538930250947603
聚类算法执行结束!
```

2）使用 nltk 分词的聚类效果：

```
In [42]: runfile('E:/codes/AnacondaCodes/Homework 3/sklearn.py'
开始数据装载:
数据装载完成!
开始执行聚类算法(使用NMI作为评价标准):
kmeans算法: 0.7993132547167876
AffinityPropagation算法: 0.7836988975391973
MeanShift算法: 0.7278222245216904
SpectralClustering算法: 0.7771086482472876
Ward hierarchical clustering算法: 0.7823244114906182
AgglomerativeClustering算法: 0.8962440814686097
DBSCAN算法: 0.737403764790242
GaussianMixture算法: 0.7988746292609659
聚类算法执行结束!
```

三.总结

实验前期主要是结合课件内容和网络资源对各个聚类算法的原理进行了系统的学习，然后是熟悉了在 python 中 sklearn 的使用，最后是进行代码编写，对各个聚类算法进行调用，并对各个算法在 Tweets 数据集上的聚类效果进行比较。实验之中也遇到了各种各样的问题，比如 sklearn 工具的首次执行必须在 c 盘下，数据要由矩阵形式转为数组形式，K 值的选取问题和不同的分词方式对聚类效果的影响等等。最后，感谢老师在这一学期对于我们的指导和帮助，最好的祝愿给您！