

# 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：去停用词，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

D:\code\AnacondaCodes\Homework 1 VSM and KNN		
名称		修
20news-18828		20
testdata		20
traindata		20
vctest		20
vctrain		20
README.txt		20
VSMandKNN.py		20
模型性能.txt		20

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}]  
...
```

]

## #预处理函数

## #读文件

```
for i in rootpathlist:
```

名;  $i$  为每一个主文件夹的路径

 $k=1;$ 

```
for j in subpathlist:
```

```
orissubspath = 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
```

#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 进行去停用词，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()

```

#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
```

```
    subpathlist = os.listdir(rootpath) #子文件夹列表
```

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

```
    for j in subpathlist:
```

```
        subpath = rootpath + os.path.sep + j
```

```
        lines = open(subpath).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 > 5:
        # 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, 即所有文档都不包含该词
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

```

#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
        subpathlist = os.listdir(rootpath) #子文件夹列表
        for j in subpathlist:
            sublist = [] #格式为: [str, dict]
            sublist.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
            subpath = rootpath + os.path.sep + j
            lines = open(subpath).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))

        sublist.append(valuesdict)          #存放 value 值为 tf-idf 的
字典 [str,dict]
        rootlist.append(sublist)
#[[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()
#KNN 的具体实现
def knn():

    k = 5

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

```

```

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 = []

        vcsun = 0 #求向量的乘积
        for key in vctest[i][1]:
            if key in vctrain[j][1]:
                vcsun = vcsun + vctest[i][1][key] *
vctrain[j][1][key]

        #求余弦 =  $AB/|A|*|B|$ 
        cosij = vcsun / ( motrain[j] * motest[i])

```

```
temp.append(vctrain[j][0])  
temp.append(cosi j)  
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)
```