Dating Mining

Experiment 1 VSM and KNN

姓名：包琛

学号：201814800

班级：2018 级计算机学硕班

指导老师：尹建华

时间：2018 年 11 月 4 日

1 实验要求

1. 预处理文本集，并且得到每个文本的 Vacter Space Model 表示
2. 实现 KNN 分类器，测试其在 20NewsGroups 的效果
3. 20%作为测试数据集，保证测试数据中各个类的文档均匀分布

数据集：http://qwone.com/~jason/20Newsgroups/

2 实验背景

2.1 Vector Space Model

Vector Space Model，向量空间模型，是把一个文本文件表示为向量的代数模型。

文档和查询都用向量来表示：

$$d_j = (w_{1,j}, w_{2,j}, \ldots, w_{t,j})$$
$$q = (w_{1,q}, w_{2,q}, \ldots, w_{t,q})$$

每一维都对应于一个个别的词组。如果某个词组出现在了文档中，那它在向量中的值就非零。已经发展出了不少的方法来计算这些值，这些值叫做（词组）权重。其中一种最为知名的方式是 tf-idf 权重。

词组的定义按不同应用而定。典型的词组就是一个单一的词、关键词、或者较长的短语。如果将词语选为词组，那么向量的维数就是词汇表中的词语个数（出现在语料库中的不同词语的个数）。

通过向量运算，可以对各文档和各查询作比较。

据文档相似度理论的假设，如要在一次关键词查询中计算各文档间的相关排序，只需比较每个文档向量和原先查询向量（跟文档向量的类型是相同的）之间的角度偏差。

实际上，我们计算向量之间夹角的余弦：

$$\cos\theta = \frac{\mathbf{d_2} \cdot \mathbf{q}}{\|\mathbf{d_2}\| \, \|\mathbf{q}\|}$$

2.2 TF-IDF

词频-逆向文档频率。

字词的重要性随着它在文件中出现的次数成正比增加，但同时会随着它在语料库中出现的频率成反比下降。

在一份给定的文件里，**词频**（term frequency，tf）指的是某一个给定的词语在该文件中出现的频率。这个数字是对**词数**（term count）的归一化，以防止它偏向长的文件。（同一个词语在长文件里可能会比短文件有更高的词数，而不管该词语重要与否。）对于在某一特定文件里的词语t_i来说，它的重要性可表示为：

$$\mathrm{tf_{i,j}} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

以上式子中$n_{i,j}$是该词在文件d_j中的出现次数，而分母则是在文件d_j中所有字词的出现次数之和。

逆向文件频率（inverse document frequency，idf）是一个词语普遍重要性的度量。某一特定词语的idf，可以由总文件数目除以包含该词语之文件的数目，再将得到的商取以10为底的对数得到：

$$\mathrm{idf_i} = \lg \frac{|D|}{|\{j : t_i \in d_j\}|}$$

其中

- |D|：语料库中的文件总数
- $|\{j : t_i \in d_j\}|$：包含词语t_i的文件数目（即$n_{i,j} \neq 0$的文件数目）如果词语不在数据中，就导致分母为零，因此一般情况下使用$1 + |\{j : t_i \in d_j\}|$

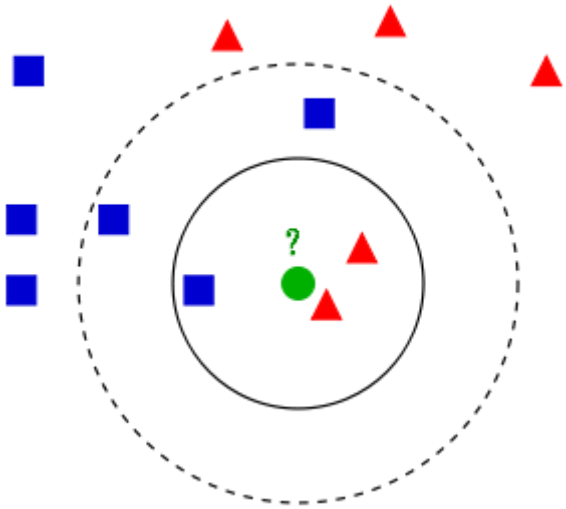
然后

$$\mathrm{tfidf_{i,j}} = \mathrm{tf_{i,j}} \times \mathrm{idf_i}$$

某一特定文件内的高词语频率，以及该词语在整个文件集合中的低文件频率，可以产生出高权重的tf-idf。因此，tf-idf倾向于过滤掉常见的词语，保留重要的词语。

2.3 KNN

K 个最近邻居法。一个对象的分类是由其邻居的“多数表决”确定的，k 个最近邻居（k 为正整数，通常较小）中最常见的分类决定了赋予该对象的类别。



k 近邻算法例子。测试样本（绿色圆形）应归入要么是第一类的蓝色方形或是第二类的红色三角形。如果 k=3（实线圆圈）它被分配给第二类，因为有 2 个三角形和只有 1 个正方形在内侧圆圈之内。如果 k=5（虚线圆圈）它被分配到第一类（3 个正方形与 2 个三角形在外侧圆圈之内）

3 实验内容

3.1 实验流程

数据集包含 20 个类的共 18828 篇文档。

1. 首先，要分别取每个类的 80%的文档作为训练集，每个类的 20%作为测试集，构建每篇文档的文档向量。
   1） 构建向量之前要对文档进行预处理，预处理之后可以得到每篇文档的关键词。我使用了 NLTK 这个 Python 库。预处理主要进行了以下几步：
      (a)分句并分词
      (b)标注词性标签，且只保留名词（标签为 NN 的词）
      (c)全部转化为小写字母
      (d)去掉标点，特殊符号和数字
      (e)词干还原
   2） 对于训练集和测试集计算并存储 tf 值。注意每个词在每篇文档中都有一个 tf 值。
   3） 对于训练集创建并存储词典。使用训练集中的关键词构建词典。为了避免词典过大，在此我过滤掉了文档频率在 15 及以下的低频词。这是合理的，因为文档频率非常低的时候，这个词是不具备区分文档类别的能力的。
   4） 对于训练集和测试集计算并存储 idf 值。对于训练集，对于上一步创建的词典中的每个词，统计出现该词的训练集文档的数目，因此词典中的每个词都有一个 idf 值。注意测试集也要进行相同的操作。
   5） 对于训练集和测试集计算 tf x idf 的值并存储在文档向量矩阵中。对于训练集，此时的文档向量是依据之前创建的词典构建的，向量的长度--也就是之后

的矩阵维度--就是词典的长度。矩阵长度就是文档个数。只需要计算词典中这些词的 tf x idf 值。注意，对于测试集，仍然使用训练集的词典构建文档向量，计算 tf x idf 值时从之前得到的测试集的 tf 值和 idf 值对应的单词中寻找和词典中的词重合的，没有在词典中出现的词就不考虑了。

6） 向量归一化。将所有的文档向量转化为单位向量。将这一步单独分离出来是为了减少后面 KNN 算法运算的时间。

2. KNN 算法，计算每个测试文档向量与训练向量的 cos 夹角值，从大到小排序，在前 K 个训练向量中选出出现次数最多的类。
3. 将测试集的真实类别与算法测出的类别相比较，计算正确率。

3.2 实验环境

处理器：Intel(R) Core(TM) i7-8700K CPU @ 3.70Ghz 3.70Ghz
RAM：16.0 GB
系统： Windows 10，64 位
编程语言：Python 3.7
IDE: PyCharm

3.2 核心代码及注释

1. 预处理

```
#stopwords 停用词列表
from nltk.corpus import stopwords
stop=stopwords.words('english')

#the stemming method 词干还原方法
from nltk.stem import WordNetLemmatizer
lemma=WordNetLemmatizer()

#the tokenization method 分词
def splitIntoWords(article):
    documentWords=[]
    tokenizer=nltk.data.load('english.pickle')
    for paragraph in article:
        sentences = tokenizer.tokenize(paragraph) #first split into
sentences

        for sentence in sentences:
            words=nltk.word_tokenize(sentence) #then split each
sentence into words
            documentWords+=words
    return documentWords
```

```
# the pre-processing method 预处理
def preProcessing(document):
    processedDocument=[]
    iter_d=iter(document)
    for word in iter_d:
        #lowerWord=word
        lowerWord=word.lower()# change to lowercase 变为小写
        word=""
        for char in lowerWord:# delete punctuation #delete
numbers 删除数字和标点符号
            if (char not in string.punctuation) and (char not in
string.digits):
                word+=char
        lowerWord=word
        #lowerWord=lowerWord.translate(string.punctuation)
        #lowerWord=lowerWord.translate(string.digits)
        #print(lowerWord)
        if len(lowerWord)==0: #the word is deleted because it only
contains punctuation or numbers
            continue
        else:
            lowerWord = lemma.lemmatize(lowerWord)   #
stemming
        if lowerWord in stop: # delete stop words
            continue
        else:
            processedDocument.append(lowerWord)
    return processedDocument
```

调用：

```
wordList=splitIntoWords(currentDoc.readlines()) #tokenization

text = nltk.Text(wordList)
tags = nltk.pos_tag(text)
wordVec = []
for tag in tags:
    if "NN" in tag[1]: # 只保留名词
        wordVec.append(tag[0])
#print(wordVec)

#print(wordList)
processedDocument=preProcessing(wordVec) #preprocessing
```

2. 计算 tf

```
tf={}
for word in processedDocument:
    if word not in tf:
        currentWordFre = count[word]
        tf[word]=currentWordFre/documentLength #tf value of
appeared word in one single document
    else:
        continue
```

3. 创建词典

```
# create a dictionary
def createDic(fileList):
    wordDic={}
    for file in fileList:
        for document in file:
            words=Counter(document[1:len(document)])
            for word in words:
                if word not in wordDic:
                    wordDic[word]=1
                else:
                    wordDic[word]+=1
    # filter those words whose document frequency is lower than 6
    for word in list(wordDic.keys()):
        if wordDic[word] <= 15:
            del wordDic[word]
    return wordDic
```

4. 计算 idf

```
def calIDF(document_count,dictionary):
    # computing idf value
    idf = {}
    for word in dictionary:
        idf[word] = math.log(abs(document_count) / (1 +
dictionary[word]))

    return idf
```

5. 构建文档向量矩阵

```
def constructMatrix(docContent,dictionary,tf,idf):
    documentMatrix = []
    for i in range(len(docContent)):
```

```
            # vector initialization
            documentVector=[]
            for j in range(len(dictionary)):
                documentVector.append(0)

            # calculate vector value
            for k in range(len(dictionary)):
                wordAndFre=dictionary[k].split("\t")
                word=wordAndFre[0]

                if word in docContent[i] and word in list(idf.keys()):
                    documentVector[k]=tf[i][word]*idf[word]
            #print(documentVector)
            documentVector=[docContent[i][0]]+documentVector
            #print(documentVector)
            documentMatrix.append(documentVector)
        return documentMatrix
```

6. 向量归一化

```
normVec=math.sqrt(sum([item*item for item in vec]))
vec= [item / normVec*10000 for item in vec]
```

7. KNN

```
with open(testPath, encoding='ISO-8859-1') as testf:
    for testVec in testf:
    # a testing document
        testVec=testVec.strip("\n").split()
        new_vec = []
        for item in testVec[1:]:
            new_vec.append(float(item))
        testVec = new_vec

        distanceSet = []

        with open(trainPath, encoding='ISO-8859-1') as trainf:
            for trainVec in trainf:
                trainVec=trainVec.strip("\n").split()
                #print(trainVec[0])
                #compute cosine value of two vectors
                new_vec = []
                for item in trainVec[1:]:
                    new_vec.append(float(item))
```

```
                    cosVal=sum([testVec[i]*new_vec[i] for i in
range(len(new_vec))])
                    #print(cosVal)
                    distanceSet.append((trainVec[0],cosVal))



# find the top 100 largest cosine value (top 100 nearest vectors)
        distanceSet.sort(key=takeSecond,reverse=True)
        #print(distanceSet)
        category100=[]
        for i in range(100):
            category100.append(distanceSet[i][0])
        #print(category100)
        countingRes=Counter(category100)
        #print(countingRes)
        maxCategory=max(countingRes.items(),key=lambda
x:x[1])[0] #find the category testVector belongs to
        count+=1
        print(count,maxCategory)
        #print("\n")
        categoryList.append(maxCategory)
testf.close()
```

4 实验结果

正确率 65.6%

5 分析与讨论

回顾实验过程，我觉得还有改进的余地。

1. 在过滤低频词这一步，我过滤掉了 document frequency 等于或小于 15 的单词。我感觉这是比过滤掉 word frequency 更合适的做法。二者的区别是，document frequency 过低的词说明这个词没有在很多文档中出现过，证明这个词没有区分度。而 word frequency 过低的词有可能是因为这个词是某一类独有的，仅在这一类中出现过，此时这种词反而是极具区分度的词了。
2. 在去掉 document frequency 等于或小于 15 的单词之后，训练集构建的词典的大小从最开始的几十万骤降到了 7000 个词。然而在 knn 时如果将训练集中的文档全用上，构建的矩阵还是太大了，因此我只抽取了训练集中的 3000 个文档（每类 150 个文档），构建

的训练集矩阵为 3000 x 7000。测试集大概 3700 多个文档，因此测试集的矩阵为 3700 x 7000。然而 knn 本身是一个效率不高的算法，因此处理时间仍然较长，给出一个测试文档向量的分类大概需要 10 秒。总的来说，knn 算法简单，效果还不错，但是效率太低。高维的文档向量最好还是降维后进行处理。