VSM and KNN

在构建 VSM 和实现 KNN 算法中，主要分为两部分完成，第一步是构建一个向量存储在列表中；第二步是把训练集每个文件中每个未被过滤掉的单词的 TF-IDF 值存储在矩阵（或字典）中，然后也把测试集每个文件中每个未被过滤掉的单词和单词的 TF-IDF 值存储在矩阵（或字典）中，最后距离用两个向量的 COS 值来计算，通过 KNN 算法得到结果。从原始数据中随机取出百分之 20 作为测试集，剩余百分之 80 作为训练集。

1.Construct a vector

在第一部分中，通过 Return_File_Name(Rootdir)函数返回训练集中所有的文件名，为后续遍历每个文件提供文件路径。通过 Get_str(name)函数把所有文件的内容保存在一个字符串中。然后是对这个字符串进行规范化，通过 Tokenization_Stemmer(str1)函数。

```
#规范化
def Tokenization_Stemmer(str1):

    zen = TextBlob(str1)

    zen = zen.words  #分词

    zen = zen.lemmatize() #名词单复数变原型

    zen = list(zen)
    #动词分词和动名词 变 原型
    for i in range(len(zen)):
        w = Word(zen[i])
        zen[i] = w.lemmatize("v")
    #所有单词变换为小写
    for i in range(len(zen)):
        zen[i] = zen[i].lower()
    zen = sorted(zen)
    return zen
```

在规范化中主要调用了 TextBlob 包，在 TextBlob 中的 w.lemmatize()函数对动词和名词恢复原形的处理效果特别好，远超 Stemmer()函数。

最后是构建一个向量，过滤掉词频小于五的单词和停用词，过滤掉包含非字母字符的单词，得到一个长度为 29269 的列表，最后把得到的向量写入文件中，在以后使用词典时直接读出即可。

```
#构建向量
def construct_vecter(list1):
    Vecter_list = []
    num = 1
    for i in range(len(list1) - 1):
        if list1[i] == list1[i + 1]:
            num += 1
            if i == len(list1) - 1 and num >= 5:
                Vecter_list.append(list1[i])
        else:
            if num >= 5 and list1[i].isalpha():
                Vecter_list.append(list1[i])
                num = 1
    Vecter_list = list(set(Vecter_list) ^ set(stop_word()))
    Vecter_list = sorted(Vecter_list)
    return Vecter_list
```

2. Constructing a VSM representation and KNN

在初次的尝试中，建立 VSM 是使用的矩阵，这个矩阵是一个 15667 * 29269 的矩阵，在计算机上运行时占用了大约百分之八十的内存，遍历这个矩阵写入每个文件每个单词的 TF-IDF 值需遍历每一项元素，进行 COS 值得计算时每两个向量相乘要计算大约 90000 次乘法，整个 KNN 算法要进大约行 2500 * 15667 次 COS 值的计算，这是一个庞大的计算量，虽然最后分类的正确率也有大约百分之 80，但是程序的空间和时间复杂度都很高，运行一次大约要 30 分钟。

之后尝试用字典的数据结构存储每个文件单词的 TF-IDF 值，这一优化，大大减少了对内存的需求，一个长度为 29269 的列表用一个长度大约为 300 - 500 的字典存储即可，减少了内存的需求，在对 KNN 算法的计算中，每一个 COS 值计算只需要大约 1000 - 2000 次左右的乘法，大大加快了运行时间，运行一次大约 10 分钟左右。

由矩阵存储每个文件的 TF-IDF 值优化为由字典存储后，每一个文件向量的规模由总向量的规模变成了每个文件的规模，时间和空间的性能都大大提高。本实验报告只包含了由字典的数据结构实现 Constructing a VSM representation and KNN。

(1). Constructing a VSM representation

由 Get_TF(filename)函数获得每一个文件中每个单词的 TF 值，返回一个字典，字典中 key 为单词，value 值为 key 出现次数。由 Get_Mat_TF(name)函数遍历所有文件，获得每个文件的字典表示，并把这些字典添加到一个列表中。

由 Get_Idf(Mat_tf,dic)函数得到向量中每个单词的 IDF 值，最后通过 Get_Tfidf(Mat_tf,dic_idf,name)函数得到最终的每个文件中每个单词的 TF-IDF 表示。

```
def Get_TF(filename):
    s = ""
    fopen = open(filename, 'r',errors='replace')

    for eachLine in fopen:
        s += eachLine
    fopen.close()

    List = Tokenization_Stemmer(s)
    len_list = len(List)
    a = {}
    for i in List:
        a[i] = round( List.count(i)/len_list , 5 ) #保留5位小数
    return a

#遍历所有文件，获得每个文件的字典表示，并把这些字典添加到一个链表中
def Get_Mat_TF(name):
    List = []
    for i in range(len(name)):
        List.append(Get_TF(name[i]))
    return List

def Get_Idf(Mat_tf,dic):
    a = {}
    for i in range(len(dic)):
        num = 0
        for j in range(len(Mat_tf)):
            if dic[i] in Mat_tf[j].keys():
                num += 1
        a[dic[i]] = num

    return a
```

```
def Get_Tfidf(Mat_tf,dic_idf,name):

    for i in range(len(Mat_tf)):
        for key in Mat_tf[i]:
            if key in dic_idf.keys():
                Mat_tf[i][key] = Mat_tf[i][key] * math.log(len(name) / (dic_idf[key] + 1))
            else:
                Mat_tf[i][key] = 0

    return Mat_tf
```

(2).KNN

KNN 中的距离通过两个向量之间的 COS 值来判断，COS 值越大，表示距离越近。通过 return_maxname(a1)函数返回前 K 个预测中出现次数最多的名字。如果这个名字与测试集中他所在的文件夹名一致，则预测成功。

其中核心的 Knn(mat_train_tfidf,mat_test_tfidf,name_train,name_test,k)函数，mat_train_tfidf 为训练集的 VSM representation，mat_test_tfidf 为测试集的 VSM representation，name_train 为测试集的文件名，name_test 为训练集的文件名，k 为 KNN 算法中 K 的值。

```python
#返回分类成功的的概率
def Knn(mat_train_tfidf,mat_test_tfidf,name_train,name_test,k):

    num = 0

    for i in range(len(mat_test_tfidf)):

        a = {}

        for j in range(len(mat_train_tfidf)):
            #计算测试集的向量与训练集中每个向量的COS值
            a[name_train[j]] = Cos_value(mat_test_tfidf[i],mat_train_tfidf[j])

        sort_a = sorted(a.items(),key = lambda x:x[1],reverse = True)#对字典中的Value值排序
        sort_a = sort_a[:k] #取前K个

        if return_maxname(sort_a) in name_test[i]: #预测成功
            num += 1

    return (num/len(mat_test_tfidf))
```

3. Classification result

当 k = 5 时，分类结果如图

```
矩阵完成
D:\测试集\alt.atheis 0.8914728682170543
D:\测试集\comp.graphic 0.7533333333333333
D:\测试集\comp.os.ms-windows.mis 0.7538461538461538
D:\测试集\comp.os.ms-windows.mi 0.7311827956989247
D:\测试集\comp.sys.ibm.pc.hardwar 0.7388535031847133
D:\测试集\comp.sys.mac.hardwar 0.8104575163398693
D:\测试集\comp.windows. 0.7714285714285715
D:\测试集\misc.forsal 0.5928571428571429
D:\测试集\rec.autos 0.8171428571428572
D:\测试集\rec.motorcycles 0.9252873563218391
D:\测试集\rec.sport.baseball 0.9038461538461539
D:\测试集\rec.sport.hocke 0.9464285714285714
D:\测试集\sci.cryp 0.9130434782608695
D:\测试集\sci.electronic 0.8021978021978022
D:\测试集\sci.me 0.8865979381443299
D:\测试集\sci.spac 0.8820224719101124
D:\测试集\soc.religion.christia 0.891566265060241
-------------
分类成功数 2103
分类总数 2534
0.82991318074191
```

当 k=10 时，分类结果如图

```
D:\测试集\alt.atheis 0.8527131782945736
D:\测试集\comp.graphic 0.76
D:\测试集\comp.os.ms-windows.mis 0.7230769230769231
D:\测试集\comp.os.ms-windows.mi 0.7204301075268817
D:\测试集\comp.sys.ibm.pc.hardwar 0.7197452229299363
D:\测试集\comp.sys.mac.hardwar 0.7973856209150327
D:\测试集\comp.windows. 0.7771428571428571
D:\测试集\misc.forsal 0.6071428571428571
D:\测试集\rec.autos 0.8
D:\测试集\rec.motorcycles 0.9252873563218391
D:\测试集\rec.sport.baseball 0.8782051282051282
D:\测试集\rec.sport.hocke 0.9523809523809523
D:\测试集\sci.cryp 0.906832298136646
D:\测试集\sci.electronic 0.7802197802197802
D:\测试集\sci.me 0.8865979381443299
D:\测试集\sci.spac 0.8932584269662921
D:\测试集\soc.religion.christia 0.8433734939759037
-------------
分类成功数 2081
分类总数 2534
0.8212312549329124
```

当 k =5 和 k = 10 时，分类成功率的总体趋势和总成功率相差无几。上图分界线以上是测试集中每个类分类的成功率，其中 rec.sport.hocke 这一类分类成功率最高，大约百分之 95.2，misc.forsal 类分类成功率最低，只有百分之 60.7。训练集中一共有 2534 个文件，其中 2081 个分类正确，分类成功的概率为百分之 82.1。