人工智能

> 教学班级：计科2班
>
> 专业：计算机科学
>
> 学号：19335262
>
> 姓名：张航悦

任务1：TF-IDF

一、 实验内容

1. 算法原理

- 利用编码来处理文档数据
- 编码原因：图像由多个像素点构成，像素值之间是可计算的。与图像不同，文本一般很难直接被进行计算，所以我们需要对文本进行编码。
- 文档编码表示
  - one-hot

    使用一个V维向量表示一篇文章，向量的长度V为词汇表的大小。1表示存在对应的单词在该文章中存在， 0表示不存在。one-hot表示仅考虑存在性，只取0/1两种值。**但这样就忽略了词语出现的频次所带来的影响，所以就进行了改进。**

  - TF：词频归一化后的概率

    每个文档的词频归一化后的概率 。

    $tf_{i,j} = n_{i,j} / (\sum_v * n_{v,d})$

    但是还是存在一些问题，**因为TF的方法容易收到一些常见词语的影响**，比如：的，吗，他们数量很多，可能会影响到一些文本关键主题的突出。

  - IDF：逆向文档频率

    如果这个词语在每篇文章都出现过，说明它能反应文本特性的能力不足，应该缩小它在词语向量中的权重。假设$|C_i|$表示第`i`个词在$|C|$篇文档中出现的次数，则 $idf_i = log\frac{|C|}{|C_i|+1}$

  - TF-IDF

    $TF-IDF = TF * IDF$，可以把IDF理解为TF的一个权重值。用以评估一字词对于一个文件集或一个语料库中的其中一份文件的重要程度。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。

2. 伪代码

进行TF-IDF运算的伪代码如下。

```
Procedure tfidf(dict,file_num,dictionary,record)
/*
input：dictionary是一个字典 键值为单词 用来统计各单词在文档中出现的总数和在每个文档中出现的次数
input:dict是对dictionary进行键值按字典序排序后的字典
input:file_num是输入文件semeval.txt的行数 即要计算的样例的个数
input：record是一个列表 顺序记录每个文档所包含的单词数

ouput：返回TFIDF矩阵
*/
    word_num:=len(dict) # 得到总单词个数
    result 创建存储结果的二维列表

    for i:=0 to len(dict): #遍历单词
        key:=dict[i]
        itf=math.log(file_num/(dictionary[key][0]+1),10) # idf计算公式
        for j:=0 to len(dictionary[key]) # 遍历计算该单词在各个文档中的tfidf
            val:=dictionary[key][j]
            if val!=0 then
               result[j-1][i]:=val/record[j-1]*itf
            end if

         end for

     end for

  return result

```

3. 关键代码展示

该实验代码可分为三部分来实现。

- 读取文件

  在读取文档同时计算了每个单词在各个文档中出现的次数和出现总数，只用遍历一次便可完成，效率较高

```
```

```
def read_csv():
    file=open("semeval.txt")
    lines=file.readlines() # 将每行文档内容存入一个列表中
    file_num=len(lines) # 记录文件数目
    record=[] # 创建一个列表 记录每个文档包含的单词数
    dictionary={} # 创建一个空字典 用来统计各单词在文档中出现的总数 和在每个文档中出现的次数

    for line in lines:
        this_line=line.split('\t',2)[2] # 取一行中的单词部分
        this_line.strip('\n') # 去掉末尾换行符
        word_list=this_line.split() # 分割出每一个单词
        record.append(len(word_list)) # 将文档的单词数记录在record中

        for word in word_list: # 遍历这一行包含的单词
            if word not in dictionary: # 如果该单词还未加入字典，即该单词第一次在文档中出现
                temp=[0 for i in range(file_num + 1)] # 创建一个临时列表 插入字典
                temp[0]=1 # 0号单元用于存储该单词在文档中出现的总次数
                temp[lines.index(line)+1]=1 # 更新该单词在该文档中的次数
                dictionary[word]=temp # 插入字典

            else: # 如果该单词已存在
                if dictionary[word][lines.index(line) + 1]==0: # 若该单词首次在该文档中出现
                    dictionary[word][0]= dictionary[word][0] + 1 # 更新该单词在文档中出现的总次数
                dictionary[word][lines.index(line) + 1]= dictionary[word][lines.index(line) + 1] + 1 # 更新该单词在该文档中的次数


    dict=sorted(dictionary) # 结果要求按字典序排序 对键值进行排序
    return dict,record,dictionary,file_num

```

- 计算TF-IDF矩阵

```
def tfidf(dict,file_num,dictionary,record):
    word_num=len(dict) # 得到总单词个数（去除重复）

```

```
    result=[[0 for i in range(word_num)] for i in range(file_num)] # 创建存储结果的二维列表

    for i in range(len(dict)): # 遍历单词
        key=dict[i] # 当前单词
        itf=math.log(file_num/(dictionary[key][0]+1),10) # idf计算公式

        for j in range(1, len(dictionary[key])): # 计算该单词在各个文档中的tfidf
            val=dictionary[key][j]
            if val!=0:
                result[j-1][i]=val/record[j-1]*itf # 因为从j=1开始存储第0个文件的信息 所以要j-1

    return result
```

- 写入csv

```
def write_csv(result):
    file = open("19335262_Zhanghangyue_TFIDF.txt", 'w')
    for i in range(len(result)):
        file.write(str(i + 1) + '\t')  # str函数将数字转成字符串
        for j in range(len(result[i])):
            if result[i][j] != 0: #将结果不为0的写入
                file.write(str(result[i][j]) + ' ')
        file.write('\n')
    file.close()
```

- 主函数

```
def main():
    dict,record,dictionary,file_num=read_csv()
    result=tfidf(dict,file_num, dictionary, record)
    write_csv(result)
```

二、 实验结果及分析

- 实验要求只输出非0元素，结果如下所示。

三、 思考题

1. IDF 的第二个计算公式中分母多了个1是为什么?

   因为可能出现某个单词在所有文本中都没有出现过的情况，这样会导致分母为0的情况出现，加1可以避免这种情况的出现。在文本总量很大的时候，加1对结果影响不大。

2. IDF数值有什么含义？ TF-IDF数值有什么含义?

   IDF是指逆向文档频率，如果这个词语在每篇文章都出现过，说明它能反应文本特性的能力不足，它可能只是一个常见词，则应该缩小它在词语向量中的权重。

   TF-IDF就是把TF值和IDF值相乘。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。

任务2：k-NN分类

一、 实验内容

1. 算法原理

- k-NN分类问题是预测离散值的问题
- k-NN处理分类问题的步骤
  - 将文本转换成向量形式，可采用one-hot或TF-IDF向量表示
  - 相似度计算：在所有向量中找出和它距离最近的k个向量

    计算距离的公式。当k=1的时候是曼哈顿距离，当k=2的时候是最常用的欧式距离。 $dist(X,Y)=\sqrt[k]{\sum_1^n(x_i-y_i)^k}$
  - 类别计算：取最相似的k个样本标签的众数
- one-hot和TF-IDF的实现代码虽都给出，但由于TF-IDF预测效果更为准确，后续实验均采用TF-IDF来实现k-NN算法
- 在实验中我利用numpy来实现TF-IDF矩阵的运算。我对训练集、验证集和测试集分别创建了自己的TF矩阵，但只创建了一个IDF向量存储每个单词的IDF值，即三个TF矩阵对应同一个IDF向量。后续计算只用进行IDF向量与TF矩阵的点乘即可。这样做既节省了储存空间，也利用了numpy加快了矩阵的运算。

2. 伪代码

- 整体思路

```
Proceduer knn_classification():
    从训练集、验证集和测试集分别读出单词和对应的情感标签，存在train_word,train_feeling,valid_word,valid_feeling,test_word,test_feeling中
    wordlist:=count_word(train_word,valid_word,test_word) #将训练集、验证集、测试集单词合并成一个大的单词表，已去重

 trtrain_tfidf,valid_tfidf,test_tfidf:=tf_idf(wordlist,train_word,valid_word,test_word) #获得训练集、验证集、测试集的tfidf矩阵
    /*调参过程*/
    for k:1 to 10 do
        predict_tag = knn_predict(valid_tfidf, train_tfidf, train_feeling, k)
        accuracy = cal_accuracy(predict_tag, valid_feeling)
        print(accuracy)
    end for

    /*写入结果*/
```

```
 test_predict=knn_predict(test_tfidf,train_tfidf,train_feeling,k=5)
    将预测结果test_predict写入文件

```

- k-NN分类

```
Function knn_predict(valid_met,train_met,train_feeling,k):
\* input: valid_met为验证集或测试集矩阵
          train_met为训练集矩阵
          train_feeling为训练集的情感标签信息
          k为k-NN算法k值的选定                *\

    result初始化为初始值为0，大小为文档数的列表
    for index:0 to 文档数 do
        row:=valid_met[index] #row为第index行的文本向量
        sum_list:=cal(row,train_met,n=1) #计算记录
        sum_index=np.argsort(sum_list) # 将sum_list中的元素从小到大排列，提取其对应的index
        dict初始化为空字典
        for i:0 to k do
            if dict中已经有了训练集排序后第i个文档的情感 then
                dict中该情感对应次数+1
            else
                dict中添加键值对{第i个情感,1}
            end if
        end for
        result[index]:=dict中情感的众数
    end for
    return result
```

- TFIDF矩阵计算

```
Function tfidf():
    word_num:=word_list的大小
    times:=np.zeros(word_num) #创建一个times向量，用来记录一个单词在文档中出现的总次数，用于计算idf
    train_tf:=np.zeros((len(train_word), word_num)) #创建训练集单词的tf矩阵
    valid_tf:=np.zeros((len(valid_word), word_num)) #创建验证集单词的tf矩阵
    test_tf:=np.zeros((len(test_word), word_num)) #创建测试集单词的的tf矩阵
    filenum:=三个单词集合的大小
```

```
    /*计算训练集的tf矩阵*/
    for i:0 to len(train_lines) do
        row:=train_lines[i]
        temp:=从row中分离出的单词列表
        num:=len(temp) #num为row中包含的单词数
        for word in temp:
            if 该单词在该文档中第一次出现 then
                times向量对应单词的位置+1
            end if
            对tf矩阵进行更新
         end for
     end for

    /*同理计算验证集和测试集的tf矩阵*/

    idf:=np.log10(filenum/(times+1)) # 计算得到idf向量
    train_tfidf:=idf*train_tf # 计算得到tfidf矩阵
    valid_tfidf:=idf*valid_tf
    test_tfidf:=idf*test_tf
    return train_tfidf,valid_tfidf,test_tfidf
```

- onehot矩阵计算

```
Function onehot(givenword,wordlist):
\* input: givenword 要求出onehot矩阵的单词列表
          wordlist 总单词列表          *\

    onehot初始化为 被求单词数x总单词数的二维矩阵
    for index:0 to len(givenword)) do
        temp:=该行中的单词
        for i:0 to len(wordlist) do
            if wordlist[i] in temp then # 如果单词在该文档内出现
onehot矩阵对应位置为1
                onehot[index][i]=1
             end if
         end for
    end for
    return onehot

```

- 距离计算

```
Function cal(row,train_tfidf,n):
    row:=转变为行向量
```

```
    temp:=将row进行行扩展得到的矩阵

    if n==2 then #计算欧式距离
        temp1:=temp-train_met
        sum_list:=temp1行内求和形成的数组
        sum_list:=sum_list开方后得到的数组
    end if

    if n==1 then #计算曼哈顿距离
        temp1:=temp-train_tfidf的矩阵，对应单元取绝对值
        sum_list=temp1行内求和形成的数组
    end if
    return sum_list
```

3. 关键代码展示

- k-NN分类

```
def knn_predict(valid_met,train_met,train_feeling,k):
    """

    :param valid_met: 待测集矩阵
    :param train_met: 训练集矩阵
    :param train_feeling: 训练集情感标签
    :param k: knn k值选定
    :return: 预测出的情感标签列表
    """
    result=[0 for i in range(valid_met.shape[0])] # 创建大小为待测集文件数的列表
    for index in range(0,valid_met.shape[0]): #遍历待测集矩阵的每一行
        row=valid_met[index]
        sum_list=cal(row,train_met,n=1)
        sum_index=np.argsort(sum_list) # 将sum_list中的元素从小到大排列，提取其对应的index
        dict = {}
        for i in range(k):#取k个样本
            if train_feeling[sum_index[i]] in dict:
                dict[train_feeling[sum_index[i]]]+=1
            else:
                dict[train_feeling[sum_index[i]]]=1
        result[index] = max(dict,key=lambda x: dict[x]) # 求众数
```

```
22
23      return result
```

- `TFIDF`矩阵计算

```
1  def tf_idf(wordlist,train_word,valid_word,test_word):
2      """
3
4      :param wordlist: 所有单词构成的表
5      :param train_word: 训练集单词
6      :param valid_word: 验证集单词
7      :param test_word: 测试集单词
8      :return: 三个集合的tfidf矩阵
9      """
10     word_num = len(wordlist) #计算出单词个数
11     times=np.zeros(word_num) #创建一个times向量，用来记录一个单词在
   文档中出现的总次数，用于计算idf
12     #times的下标和wordlist中单词的下标对应
13     train_tf=np.zeros((len(train_word), word_num)) #创建训练集单
   词的tf矩阵 规格为文档数x总单词数
14     valid_tf=np.zeros((len(valid_word), word_num)) #创建验证集单
   词的tf矩阵
15     test_tf=np.zeros((len(test_word), word_num)) #创建测试集单词
   的的tf矩阵
16     filenum=len(train_word)+len(valid_word)+len(test_word) #计
   算文件总数
17     for i,row in enumerate(train_word):
18         temp=row.split(' ') #将一个文档内的的单词分离出来
19         num=len(temp) #num为一个文档中的单词数
20         for word in temp:
21             index=wordlist.index(word) # 获取该单词在wordlist中的
   下标
22             if train_tf[i][index]==0:
23                 times[index]+=1  # 若该单词在该文档中第一次出现 更新
   times值
24             train_tf[i][index]=(train_tf[i][index]*num+1)/num
   #对tf值进行更新
25
26     for i,row in enumerate(valid_word):
27         temp=row.split(' ')
28         num=len(temp)
29         for word in temp:
30             index=wordlist.index(word)
31             if valid_tf[i][index]==0:
32                 times[index]+=1
```

```
            valid_tf[i][index]=(valid_tf[i][index]*num+1)/num

    for i,row in enumerate(test_word):
        temp=row.split(' ')
        num=len(temp)
        for word in temp:
            index=wordlist.index(word)
            if test_tf[i][index]==0:
                times[index]+=1
            test_tf[i][index]=(test_tf[i][index]*num+1)/num

    idf=np.log10(filenum/(times+1)) # 计算得到idf向量
    train_tfidf=idf*train_tf # 计算得到tfidf矩阵
    valid_tfidf=idf*valid_tf
    test_tfidf=idf*test_tf
    return train_tfidf,valid_tfidf,test_tfidf

```

- `onehot`矩阵计算

```
def onehot(givenword, wordlist):
    """

    :param givenword: 要求onehot矩阵的单词列表
    :param wordlist: 所有单词构成的表
    :return: 所求单词表的onehot矩阵
    """
    onehot=np.zeros(shape=(len(givenword), len(wordlist))) # 创建一个二维矩阵
    for index in range(len(givenword)):
        temp=givenword[index].split(' ') # 将一行内的单词分割出来
        for i, word in enumerate(wordlist):
            if word in temp: # 如果单词在该文档内出现 onehot矩阵对应位置为1
                onehot[index][i]=1
    return onehot

```

- 距离计算

```
def cal(row,train_met,n):
    """

    :param row: 一行单词
    :param train_met: 训练集tfidf或onehot矩阵
```

```
    :param n: 计算方式 n=1为曼哈顿距离 n=2为欧氏距离
    :return: 记录该行和训练集的距离的列表
    """
    row=row.reshape(1,-1)
    temp=np.repeat(row,train_met.shape[0],axis=0) # 将该行复制将temp扩充为一个维度和train_met维度相同的矩阵 便于计算

    if n==2: # 计算欧式距离
        temp1=np.square(temp-train_met)
        sum_list=np.sum(temp1,axis=1) #行内求和
        sum_list=np.sqrt(sum_list)

    if n==1: #计算曼哈顿距离
        temp1=np.abs(temp-train_met)
        sum_list=np.sum(temp1,axis=1)
    return sum_list
```

- 主函数

```
def main():
    train_word,train_feeling=read_csv("train_set.csv")
    valid_word,valid_feeling=read_csv("validation_set.csv")
    test_word=read_csv("test_set.csv")
    wordlist=count_word(train_word,valid_word,test_word)
    valid_onehot=onehot(valid_word,wordlist)
    train_onehot=onehot(train_word,wordlist)


 #train_tfidf,valid_tfidf,test_tfidf=tf_idf(wordlist,train_word,valid_word,test_word)

    for k in range(3,17):

 predict_tag=knn_predict(valid_onehot,train_onehot,train_feeling,k)
        accuracy=cal_accuracy(predict_tag,valid_feeling)
        print('k = ' + str(k) + ', accuracy = ' + str(accuracy))

        #predict_tag = knn_predict(valid_tfidf, train_tfidf, train_feeling, k)
        #accuracy[k-3] = cal_accuracy(predict_tag, valid_feeling)
        #print('k = ' + str(k) + ', accuracy = ' + str(accuracy[k-3]))
```

```
'''
test_predict=knn_predict(test_tfidf,train_tfidf,train_feeling,k=5)
    test_output = pd.DataFrame({'Words (split by space)': test_word, 'label':test_predict})
test_output.to_csv('19335262_Zhanghangyue_KNN_classification.csv',index=None,encoding='utf8')  # 参数index设为None则输出的文件前面不会再加上行
'''
```

4. 创新点

引入了`numpy`矩阵，，加速运算。在计算距离的过程中将test向量扩展成了test矩阵，利用`numpy`封装好的矩阵操作，一次性计算出1个test和训练集所有样例之间的距离。在计算`tfidf`矩阵的时候利用一个`idf`向量和三个`tf`矩阵进行点乘运算进而得到`tfidf`矩阵。

二、 实验结果及分析

1. 结果展示及分析

通过循环调用KNN分类算法，遍历不同的k值，就可以找到在这个模型下训练集上表现最好时候的k值和它在验证集上对应的准确度。采用TFIDF的方法，取k=3到16，依次输出精确度。

- 取曼哈顿距离时结果为

```
k = 3, accuracy = 0.41479099678456594
k = 4, accuracy = 0.40192926045016075
k = 5, accuracy = 0.40514469453376206
k = 6, accuracy = 0.3987138263665595
k = 7, accuracy = 0.40514469453376206
k = 8, accuracy = 0.3954983922829582
k = 9, accuracy = 0.3762057877813505
k = 10, accuracy = 0.3890675241157556
k = 11, accuracy = 0.40192926045016075
k = 12, accuracy = 0.40836012861736337
```

```
k = 13, accuracy = 0.41479099678456594
k = 14, accuracy = 0.3987138263665595
k = 15, accuracy = 0.40836012861736337
k = 16, accuracy = 0.40514469453376206
```

- 取欧氏距离时结果为

```
k = 3, accuracy = 0.26366559485530544
k = 4, accuracy = 0.2379421221864952
k = 5, accuracy = 0.21543408360128619
k = 6, accuracy = 0.2090032154340836
k = 7, accuracy = 0.19614147909967847
k = 8, accuracy = 0.1864951768488746
k = 9, accuracy = 0.17363344051446947
k = 10, accuracy = 0.17684887459807075
k = 11, accuracy = 0.1864951768488746
k = 12, accuracy = 0.19614147909967847
k = 13, accuracy = 0.2057877813504823
k = 14, accuracy = 0.22186495176848875
k = 15, accuracy = 0.22508038585209003
k = 16, accuracy = 0.2347266881028939
```

- 可能是由于训练及验证样本数量比较少，可以看到准确度随着k值总体在小范围内摆动，变化没有一个很明显的变化趋势，
- 取曼哈顿距离计算时，准确率明显高于取欧式距离。在k=3和13时，都达到了41.479%的准确率。

2. 模型性能展示和分析

对比采用`onehot`和`TFIDF`编码方式，取曼哈顿和欧式距离时k值和准确率。

	曼哈顿距离	欧式距离	onehot	TFIDF	准确率	k
初始	1	0	0	1	41.497%	3或13
优化1	0	1	0	1	23.367%	3
优化2	0	1	1	0	40.836%	13
优化3	1	0	1	0	40.836%	13
最终	1	0	0	1	41.497%	3或13

看到这个结果的时候我也觉得很神奇，欧式距离和曼哈顿距离的onehot编码方式在同一k值取得同一准确率（我debug了一下，的确不是bug的问题）。

可以看到在短文本情况下，`onehot`编码方式的平均下来，在短文本的数据集上倾向比`TFIDF`编码方式的表现更好，采用曼哈顿距离表现比欧式距离效果更好。

任务3：k-NN回归

一、 实验内容

1. 算法原理

- k-NN回归问题是预测连续值的问题
- k-NN处理回归问题的步骤
  - 将文本转换成向量形式，可采用one-hot或TF-IDF向量表示
  - 相似度计算：在所有向量中找出和它距离最近的k个向量
  - 根据相似度加权计算：将距离最近的k个向量的距离的倒数作为权重，计算test属于该标签的概率（以happy为例）

$$P(test\ y\ is\ happy) = \sum_{x=1}^{n} \frac{train\ x\ probability}{d(train\ x, test\ y)}$$

  - 在回归问题的应用景下，相关系数用于计算实际概率向量以及预测概率向量之间的相似性

$$COR(X,Y) = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{\sum_{i=1}^{n}(X_i-\bar{X})(Y_i-\bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i-\bar{X})^2 \sum_{i=1}^{n}(Y_i-\bar{Y})^2}}$$

  - 分别计算6个情感的真实概率值和预测概率值的相关系数，然后对6个维度取平均值计算得到最终的相关系数，评价模型。

2. 伪代码

k-NN回归的代码与k-NN分类的代码总体相同，主要区别在于读入函数和k-NN的标签判断的函数。相同部分不再赘述。

- 读入函数

```
Function read_csv(file_name):
    打开文件，跳过表头开始读取
    word新建空列表 #存储单词
    feeling新建一个情感标签数x文档数的二维列表
    #0-5行分别代表对应的6个情感，每一列代表一个文档的结果
```

```
    if 读取的文件是训练集或验证集：
        for row in reader:
            在word中添加该行单词
            for j:0 to 5 do
                在feeling的对应行添加信息
            end for
        end for
    end if
    return word,feeling

    if 读取的文件是测试集：
       for row in reader:
           在word中添加该行单词
       end for
    end if

    return word
```

- k-NN回归预测函数

  为保证六个情感标签概率之和为1，要记得做归一化处理。

```
Function knn_predict(valid_met,train_met,train_feeling,k):
\* input: valid_met为验证集或测试集矩阵
          train_met为训练集矩阵
          train_feeling为训练集的情感标签信息
          k为k-NN算法k值的选定          *\

    feeling_num:=len(train_feeling)
    新建result 为便于后续写入csv文件 将result存为情感数x文件数的二维列表

    for index：0 to valid_met.shape[0] do
        row:=valid_met[index]
        sum_list:=cal(row,train_met,n=1)
        sum_index:=np.argsort(sum_list)
        total:=0
        for i:0 to k do
            for j:0 to 5 do
                if sum_list[sum_index[i]]==0:
                    sum_list[sum_index[i]]=0.001 # 若这两个标签完全相同，则将距离改为0.001 避免分母为0的情况出现
                result[j][index]+=train_feeling[j][sum_index[i]]/float(sum_list[sum_index[i]]) # 根据公式计算概率
```

```
                total+=train_feeling[j]
[sum_index[i]]/float(sum_list[sum_index[i]]) # 计算总和 用于做归一化处理
            end for
        end for

        for i:0 to 5 do
            result[i][index]/=total #归一化处理
        end for
    end for
    return result
```

3. 关键代码

- 读入函数

```
def read_csv(file_name):
    """

    :param file_name: 要读入的csv文件名
    :return:返回单词列表 情感标签列表
    """

    with open(file_name,'r') as f:
        reader=csv.reader(f)
        first_row=next(reader)
        feeling_num=len(first_row)-1

        word=[] #存储单词
        feeling=[[] for i in range(feeling_num)] #创建一个情感标签数x文档数的二维列表
        #第0-5行分别代表对应的5个情感

        if file_name=="train_set1.csv" or file_name=="validation_set1.csv":
            for row in reader:
                word.append(row[0])
                for j in range(feeling_num):
                    feeling[j].append(float(row[j+1])) #一个文档的情感标签是一列 注意要转成float 不然无法运算

            return word,feeling
        else:
```

```
            for row in reader:
                word.append(row[1])
            return word
```

- k-NN回归函数

```
def knn_predict(valid_met,train_met,train_feeling,k):
    feeling_num=len(train_feeling)
    result=[[0.0 for i in range(valid_met.shape[0])] for j in range(feeling_num)]
    # 为便于后续写入csv文件 将result列表存为情感数x文件数的规格

    for index in range(0,valid_met.shape[0]):
        row=valid_met[index]
        sum_list=cal(row,train_met,n=1)
        sum_index=np.argsort(sum_list)
        total=0
        for i in range(k):
            for j in range(feeling_num):
                if sum_list[sum_index[i]]==0:
                    sum_list[sum_index[i]]=0.001 # 若这两个标签完全相同，则将距离改为0.001 避免分母为0的情况出现
                result[j][index]+=train_feeling[j][sum_index[i]]/float(sum_list[sum_index[i]]) # 根据公式计算概率
                total+=train_feeling[j][sum_index[i]]/float(sum_list[sum_index[i]]) # 计算总和 用于做归一化处理

        for i in range(feeling_num):
            result[i][index]/=total #归一化处理

    return result
```

二、 实验结果及分析

1. 结果展示及分析

通过循环调用KNN回归算法，遍历不同的k值，就可以找到在这个模型下训练集上表现最好时候的k值和它在验证集上对应的相关系数。采用TFIDF的方法，取k=3到16，依次输出相关系数。相关系数利用提供的`validation相关度评估.xlsx`完成计算

- 取曼哈顿距离时结果为

```
k=3, coefficient=0.31635738697656807
k=4, coefficient=0.32525132555084196
k=5, coefficient=0.3227160899046577
k=6, coefficient=0.32522254267169204
k=7, coefficient=0.31211085366060115
k=8, coefficient=0.316372564257192
k=9, coefficient=0.30785183237412894
k=10, coefficient=0.31469633558551763
k=11, coefficient=0.32209775996910595
k=12, coefficient=0.32451177938791037
k=13, coefficient=0.3269034632830868
k=14, coefficient=0.33245083187909436
k=15, coefficient=0.32580554764405917
k=16, coefficient=0.326640413606699
k=17, coefficient=0.3264130073165443
k=18, coefficient=0.32156733955725253
k=19, coefficient=0.32148629511863946
k=20, coefficient=0.32196083321168806
```

k=13时，相关程度最高，相关系数为0.3269034632830868

- 取欧式距离时结果为

```
k=3, coefficient=0.2584148871442702
k=4, coefficient=0.25389903943289444
k=5, coefficient=0.25701020049336293
k=6, coefficient=0.23673054270264302
k=7, coefficient=0.24448088405709925
k=8, coefficient=0.24896973852120216
k=9, coefficient=0.24264594129196726
k=10, coefficient=0.2386779279005535
k=11, coefficient=0.24944531158353658
k=12, coefficient=0.24757271951825477
k=13, coefficient=0.24995861533584782
k=14, coefficient=0.25392654817599386
k=15, coefficient=0.24920159446999635
k=16, coefficient=0.24917490031039183
k=17, coefficient=0.2602504488449418
k=18, coefficient=0.2541627393655171
k=19, coefficient=0.25134492256685137
k=20, coefficient=0.24169062856307888
```

k=17时，相关程度最高，相关系数为0.2602504488449418

2. 模型性能展示和分析

	曼哈顿距离	欧式距离	onehot	TFIDF	相关系数	k
初始	1	0	0	1	0.3269	13
优化1	0	1	0	1	0.2603	17
优化2	0	1	1	0	0.2699	10
优化3	1	0	1	0	0.2798	10
最终	1	0	0	1	0.3269	13

在分类和回归问题上，都是TFIDF编码方式结合曼哈顿距离的效果最好。结果与分类问题类似。

三、 思考题

为什么是倒数？如果要求同一测试样本的各个情感概率总和为1，应该如何处理？

１．因为距离越远理论上和当前文本的相似程度就越小，则它作为权重削减它的情感对当前文本情感的影响力，所以需要取倒数，分母越大，数越小。

２． 归一化处理：将所有情感概率加起来的到一个total，然后每个情感概率除以这个total作为归一化后的情感概率。