人工智能实验lab1 实验报告

> 学号：
>
> 姓名：TRY
>
> 专业：计算机科学与技术

Task 1：TFIDF

1. 算法原理

- **文本处理**包括4大步骤：分词、去停用词、建立词表、对文档的词语进行编码。其中，本次实验的文档已将前三步处理好，只需要我们对文档的词语进行编码。
- 而文本数据处理（也就是"编码"）有以下几种方式：
  - **One-hot编码**：每一个词是一个V维的向量（V是词表大小）
  - **词频表示**：每一个文档都是一个V维的向量，其中每一维的值对应词表中该词语出现的次数
  - **词频归一化后的概率表示（tf）**：每个文档的词频归一化后的概率
    - $$tf_{i,d} = \frac{n_{i,d}}{\sum_v n_{v,d}}$$
  - **逆向文档频率（idf）**：表示词语普遍重要性的度量。
    - 假设$|C_i|$表示第`i`个词在$|C|$篇文档中出现的次数，则
    $$idf_i = log\frac{|C|}{|C_i|}$$
  - **TF-IDF**：将文本转化成向量矩阵，TF*IDF表示：
    - 假设$|D_i|$表示第`i`个词在$|D|$篇文档中出现的次数，则
    $$tf\_idf_{i,j} = \frac{n_{i,j}}{\sum_{k=1}^{V} n_{i,k}} * lg\frac{|D|}{1+|D_j|}$$
    - **注**：由于一个词可能在该集合中没有出现，即$D_j = 0$，因此这里需要将分母进行$+1$操作来避免分母=0的情况。
- **上面5种方法的区别与联系**：由于**onehot**忽略了词语出现的频率的影响，所以引入了TF方法；然而**TF方法**可能会收集到许多常用词语，如一些停用词，它们的频次很高但却没有实际意义，反而会影响其他关键词语的比例，所以引入了idf方法；**idf方法**亦称为反文档频率，当这个词语在每个文档都出现过时，表示它能反应文本特性的能力不足，应降低他们在词语向量中的权重；最终，形成了**TF-IDF方法**，将TF矩阵与IDF矩阵相乘，来表示文本。
- **而笔者的`tf-idf`矩阵的设计思路如下：**
  - 在task1中，笔者使用了字典**dictionary**来表示`tf-idf`矩阵。其中，第一维的大小（即行数）是文档中单词的个数，第二维的大小（即列数）是文档个数+1。其中第一列也就是$dic[x][0]$是用来存储该单词在所有文档中出现的次数$|D_j|$，后面的第1~n列是用来存储该单词在第1~n个文档中出现的频次$n_{i,j}$。
  - 并利用另一个list变量**word_num**来储存每个文档的单词数。

- 这样的**好处**在于，只需要**遍历一遍文档**就可以统计出数据，构建`tf-idf`矩阵的初始版本，之后在计算`tf-idf`矩阵相乘的时候，再将初始版本的`dic`字典进行处理（将第1~n列的数据除以各自文档的单词数可以得到`tf`矩阵，利用第0列的数据可以构造得到`idf`矩阵），再相乘就可得到结果。
- *注：由于这是我第一次写python程序，故仍利用了dictionary来表示矩阵。然而，该表示方法在task2中利用numpy矩阵进行了改进。*

2. 伪代码

```
Procedure TFIDF()
    lines:= 读文件的内容
    file_num := len(lines)
    dictionary字典，初始化为空
    word_num列表，各单元初始化为0
    for line in lines:
        word_temp := 用split分隔line得到的单词list
        更新word_num，将line的单词数存储到对应单元
        for word in word_temp:
            if word在dictionary中 then
                temp_list := 大小为file_num+1的全0列表
                temp_list[0] := 1
                temp_list[lines.indes(line)+1] := 1
                将{word, temp_list}键值对插入到dictionary中
            else
                if word在line这个文档中没有出现过 then
                    dictionary中word对应的列表的[0]位置的值+1
                end if
                dictionary中word对应的列表的对应列的值+1
            end if
        end for
    end for
    dic_order := dictionary的升序排序
    result二维列表，各单元初始化为0
    for i:=0 to len(dic_order):
        key := dic_order[i]
        idf := 0
        for j:=0 to len(dictionary[key]):
            val := dictionary[key][j]
            if j=0 then
                idf := log_10(file_num/(val + 1))
            else
                if val!=0 then
                    result[j-1][i] := val/(word_num[j-1])*idf
                end if
            end if
        end for
    end for
    将result列表输出到文件中
```

3. 代码展示

- 由于task 1是初次接触python，所以未使用模块化编程。这将会在task 2得到改善。

- 代码如下：

```
import os
import math

f = open("semeval.txt")
lines = f.readlines()        # 将所有的行一次性读出来，这样可以知道总文档数
file_num = len(lines)    # num是文档数（即文件的行数）
dictionary = {}
word_num = [0 for i in range(file_num)]       # word_num列表存储各行中的单词个数
for line in lines:    # 按行读取文件
    str1 = line.split('\t', 2)[2]      # 按照\t来分隔出最后的单词，且取第二个单元
    str1 = str1.strip('\n')      # 去掉行末的换行符
    word_temp = str1.split(' ')  # 也可不加参数，默认为空格

    word_num[lines.index(line)] = len(word_temp)

    for word in word_temp:
        if word not in dictionary:   # 如果字典里面没有这个单词(without "()")
            temp_list = [0 for i in range(file_num+1)]      # 创建一个长度为num+1的临时列表
            temp_list[0] = 1    # 第0单元代表该单词在文件中的多少个文档中出现过
            temp_list[lines.index(line)+1] = 1    # 在列表的对应项置为1，代表在对应的文档中有1个这个单词
            dictionary[word] = temp_list    # 将这个键值对插入到字典中
        else:       # 如果字典里面有这个单词
            if dictionary[word][lines.index(line)+1] == 0:  # 原来这一行前面没有这个单词
                dictionary[word][0] = dictionary[word][0]+1      # 文档数+1
            dictionary[word][lines.index(line)+1] = 1 + dictionary[word][lines.index(line)+1]   # 无论原来这一行前面有没有这个单词，数量都加1
# 排序：
dic_order = sorted(dictionary)       # ！！用这个方法，这样就不会转成元组，只需要知道键值排序就好！然后再去访问原来的字典去访问对应的列表.

distinct_word_num = len(dictionary)       # distinct_word_num存储的是文件中不同的单词个数
result = [[0 for i in range(distinct_word_num)]for i in range(file_num)]
 # 结果二维矩阵

for i in range(len(dic_order)):
    key = dic_order[i]       # 获得了键值：也就是单词
    idf = 0
    for j in range(len(dictionary[key])):   # 遍历每个单词对应的列表
        val = dictionary[key][j]
        if j == 0:
            idf = math.log(file_num/(val+1), 10)       # 求出对应单词的idf
        else:
            if val != 0:  # 只对不为0的进行处理
                result[j-1][i] = val/(word_num[j-1])*idf

file = open('18340159_tangry_TFIDF.txt','w')
for i in range(len(result)):
    file.write(str(i)+'\t')      # str函数将数字转成字符串
    s = str(result[i])+'\n'
    file.write(s)
file.close()
```

4. 实验结果与分析

- TF-IDF实验要求是通过输入文本计算每个文本的词向量的TF-IDF表示，并且词语要按照单词升序排序。 结果如下：

```
file_num a abba abduct abductor abdul abil abort about abus accept accessori accident accord accumul accus acid acknowledg across act action acti
1   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
2   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
3   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
4   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
5   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
6   [0.20340946315524178, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
7   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
8   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
9   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
10  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
11  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
12  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
13  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
14  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
15  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
16  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
17  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.18434984907593321, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
18  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
19  [0.20340946315524178, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
20  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
21  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
22  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
23  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
24  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

- 由于整个文件中的单词数量非常多，而每个文档的单词数有限，所以取0的值非常多。且观察可知，在第6个文档中，确实有单词“a”（图片如下），而在输出文档中，第6行的第一个元素的确不为0，因此正确。其他文档的结果亦可对应分析

```
6   all:62 anger:0 disgust:0 fear:0 joy:34 sad:0 surprise:28    pm havana deal a good experi
```

Task 2：KNN分类

1. 算法原理

- KNN分类问题是预测离散值的问题，例如本题中的预测情感标签。
- 具体步骤： 通过将各个文本转化成向量形式，然后在所有向量中找出和它最相似的决定情感标签。其中，“相似”由距离来衡量，即找出距离最近的k个向量，并分别统计这k个向量的标签，找到最多相同的标签（即众数），用它作为目标文本的标签值 。
- 这里涉及了距离的衡量。其中，数学中定义了L_p距离：

$$L_p(x_i, x_j) = (\sum_{l=1}^{n} |x_i^{(l)} - x_j^{(l)}|^p)^{\frac{1}{p}}$$

  - 当P=1时，为曼哈顿距离：

$$L_1(x_i, x_j) = \sum_{l=1}^{n} |x_i^{(l)} - x_j^{(l)}|$$

  - 当P=2时，为欧氏距离：

$$L_2(x_i, x_j) = (\sum_{l=1}^{n} |x_i^{(l)} - x_j^{(l)}|^2)^{\frac{1}{2}}$$

- 将文本转化成向量形式有很多方法，如onehot、tf、或TF-IDF。以下采用TF-IDF方法进行。
- 其中，为了方便矩阵运算，笔者将train\validation\test三个集合的词语放在了一起，形成了一个**大词表**。其中，三个集合对应的`idf`矩阵是相同的，只不过各自的`tf`矩阵不同。这样的好处是：可以通过**numpy运算加快**计算速度。

- numpy库是python中拥有丰富的数组计算函数的库。numpy的数组元素是连续存储的，且在底层会进行向量化处理，因此与list相比，numpy数组的计算效率大大提高。且numpy中封装了很多函数，方便使用。

2. 伪代码

```
Procedure KNN_classification()
    从训练集、验证集、测试集中分别读出train_lines, train_labels, valid_lines, valid_labels, test_lines
    word_list := call count_words(train_lines, valid_lines, test_lines) /*统计去重后的大词表，包括三个集合的词语*/
    train_tfidf, valid_tfidf, test_tfidf = call tf_idf(train_lines, valid_lines, test_lines, word_list) /*获得三个集合的tf-idf矩阵*/
    /*调参过程*/
    k := 3
    while k<20 do
        valid_predict := call KNN_predict(train_tfidf, valid_tfidf, k, train_labels)
        accuracy := call calculate_accuracy(valid_labels, valid_predict)
        print(准确率)
        k+=1
    end
    /*输出测试结果*/
    test_predict := call KNN_predict(train_tfidf, test_tfidf, 13, train_labels)
    输出test_predict到csv文档
```

```
Function KNN_predict(train_tfidf, valid_tfidf, k, train_labels)
    result初始化：大小为文档数，初值为0的列表
    for index in range(0,文档数)
        row := 验证集/测试集中第index行的文档向量
        sum_list := call calculate_distance(train_tfidf, row, False)    /*测量距离*/
        sort_index := np.argsort(sum_list)
        dict1初始化为空
        i := 0
        while i<k do
            if dict1中已经有了训练集排序后第i个文档的情感 then
                dict1对应情感的值+1
            else
                dict1中添加键值对{第i个情感,1}
            end if
        result[index] := 字典dict1中情感的众数
        end
    end for
    return result
```

```
Function calculate_distance(train_tfidf, row, flag)
    row := 转成行向量
    temp := 用row进行行扩展形成的矩阵
    if flag = True then /*曼哈顿距离*/
        temp1 := temp - train_tfidf的矩阵，并对应单元取绝对值
        sum_list := temp1行内求和形成的数组
    else    /*欧氏距离*/
        temp1 := temp - train_tfidf的矩阵，并对应单元取平方
        sum_list := temp1行内求和形成的数组
        sum_list := sum_list对应单元开方的数组
    end if
    return sum_list
```

```
/*numpy矩阵版本的tf_idf相乘*/
Function tf_idf(train_lines, valid_lines, test_lines, word_list)
    word_num := word_list大小
    file_num := 三个集合的大小之和
    idf := np.zeros(word_num, dtype=float)  /*利用numpy.zeros创建全零的numpy向量*/
    train_idf := np.zeros((len(train_lines), word_num), dtype=float)    /*利用
numpy.zeros建立二维矩阵*/
    valid_tf := np.zeros((len(valid_lines), word_num), dtype=float) /*利用
numpy.zeros建立二维矩阵*/
    test_tf := np.zeros((len(test_lines), word_num), dtype=float)   /*利用
numpy.zeros建立二维矩阵*/
    /*建立train的tf矩阵，同时更新if向量*/
    for i:=0 to len(train_lines):
        row := train_lines[i]
        word_temp := row取出单词的列表
        total = word_temp的单词数
        for word in word_temp:
            if 本文档之前没有这个单词 then
                idf对应单词的位置+1
            end if
            更新train_tf的值
        end for
    end for
    /*建立validation的tf矩阵，同时更新if向量。（过程同上，不再赘述）*/
    /*建立test的tf矩阵，同时更新if向量。（过程同上，不再赘述）*/
    idf := np.log10(file_num/(idf+1))
    train_tfidf := idf * train_tf   /*numpy矩阵相乘：对应位置相乘*/
    valid_tfidf := idf * valid_tf
    test_tfidf := idf * test_tf
    return train_tfidf, valid_tfidf, test_tfidf
```

3. 代码展示

- 读文件板块：

```
def read_file_csv(file_name):
    """
    :param file_name: 读取train_set,validation_set,test_set三个文件的内容(如果是
前两者，则将情感也读出来)
    :return: 返回文件内容&情感，即文件的第0列和第1列
    """
```

```
    with open(file_name, 'r') as f:
        reader = csv.reader(f)
        labels = []
        lines = []
        if file_name == 'train_set.csv' or file_name ==
'validation_set.csv':
            for row in reader:
                labels.append(row[1])
                lines.append(row[0])
            del(lines[0])   # 由于reader没有index函数，所以就只能在构建完list之后
再删除第一行
            del(labels[0])
            return lines, labels
        else:
            for row in reader:
                lines.append(row[1])
            del (lines[0])
            return lines
```

- 利用`list`收集`train_lines`，`validation_lines`，`test_lines`里面的词语，且利用`set`去重，**加速去重过程。**

```
def count_words(train_lines, valid_lines, test_lines):
    """
    :param train_lines: 训练集的词表
    :param valid_lines: 验证集的词表
    :param test_lines: 测试集的词表
    :return: 去重后的大词表list
    """
    word_set = set()    # 使用set来进行查找，快!
    word_list = []
    for row in train_lines:
        word_temp = row.split(' ')
        for i in word_temp:
            if i not in word_set:
                word_set.add(i)
                word_list.append(i)
    for row in valid_lines:
        word_temp = row.split(' ')
        for i in word_temp:
            if i not in word_set:
                word_set.add(i)
                word_list.append(i)
    for row in test_lines:
        word_temp = row.split(' ')
        for i in word_temp:
            if i not in word_set:
                word_set.add(i)
                word_list.append(i)
    return word_list
```

- 计算`tf_idf`矩阵：利用了numpy矩阵来表示idf矩阵，并利用numpy中的`“*”`操作来处理tf和idf矩阵的相乘。

```
def tf_idf(train_lines, valid_lines, test_lines, word_list):
    """
```

```
    这样子只用遍历一遍，即可同时得到tf和idf矩阵，并通过点乘求得tf-idf矩阵
    :param train_lines: 训练集文本
    :param valid_lines: 验证集文本
    :param test_lines: 测试集文本
    :param word_list: 去重后的大单词表
    :return: 三者的TF-IDF矩阵
    """
    word_num = len(word_list)
    file_num = len(train_lines) + len(valid_lines) + len(test_lines)
    idf = np.zeros(word_num, dtype=float)    # 创建一维idf数组，大小为word_list
长度，初始化为0
    train_tf = np.zeros((len(train_lines), word_num), dtype=float)     # 创
建三种二维的tf矩阵，初始化为0
    valid_tf = np.zeros((len(valid_lines), word_num), dtype=float)
    test_tf = np.zeros((len(test_lines), word_num), dtype=float)
    for i, row in enumerate(train_lines):
        word_temp = row.split(' ')
        total = len(word_temp)
        for word in word_temp:
            index = word_list.index(word)
            if train_tf[i][index] == 0:     # 更新idf矩阵
                idf[index] += 1
            train_tf[i][index] = (train_tf[i][index]*total+1)/total
    for i, row in enumerate(valid_lines):
        word_temp = row.split(' ')
        total = len(word_temp)
        for word in word_temp:
            index = word_list.index(word)
            if valid_tf[i][index] == 0:     # 更新idf矩阵
                idf[index] += 1
            valid_tf[i][index] = (valid_tf[i][index]*total+1)/total
    for i, row in enumerate(test_lines):
        word_temp = row.split(' ')
        total = len(word_temp)
        for word in word_temp:
            index = word_list.index(word)
            if test_tf[i][index] == 0:     # 更新idf矩阵
                idf[index] += 1
            test_tf[i][index] = (test_tf[i][index]*total+1)/total
    idf = np.log10(file_num/(idf+1))
    # idf1 = np.repeat(idf, len(train_lines),axis=0)      # 扩展行数，使得可以
使用
    train_tfidf = idf * train_tf    # 利用点乘操作
    valid_tfidf = idf * valid_tf
    test_tfidf = idf * test_tf
    return train_tfidf, valid_tfidf, test_tfidf
```

- 计算1个test和训练集各样例之间的距离：这里采用了numpy进行扩展，即将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test和训练集所有样例之间的距离。（这也是为什么要将三个集合的词语放到大词表中，方便了距离的计算）并且，这里有flag参数，可以决定距离是曼哈顿距离还是欧氏距离。

```
def calculate_distance(train_tfidf, row, flag):
    """
    这里，运用了numpy进行举例的计算，一次性将1个test和所有的训练集进行了距离计算，加
速！！
    :param train_tfidf: 训练集
```

```
    :param row: 验证集
    :param flag: 表示用的是什么距离
    :return: 距离的list
    """
    row = row.reshape(1, -1)      # ？？row自己变成了列向量，要转回成行向量
    temp = np.repeat(row, train_tfidf.shape[0], axis=0)    # 将行向量扩展成矩阵
    if flag is True:  # 曼哈顿距离
        temp1 = np.abs(temp - train_tfidf)  # np求相减之后的绝对值
        sum_list = np.sum(temp1, axis=1)  # 行内求和
    else:  # 欧氏距离
        temp1 = np.square(temp - train_tfidf)  # np求相减之后的平方
        sum_list = np.sum(temp1, axis=1)  # 行内求和
        sum_list = np.sqrt(sum_list)
    return sum_list
```

- KNN_predict函数，预测测试样本/验证样本的label：

```
def KNN_predict(train_tfidf, valid_tfidf, k, train_labels):
    """
    :param train_tfidf: 训练集
    :param valid_tfidf: 验证集
    :param k: k值
    :param train_labels: 训练集的情感集合
    :return: 返回预测的情感集合
    """
    result = [0 for i in range(valid_tfidf.shape[0])]
    for index in range(0, valid_tfidf.shape[0]):      # ！！按照下标来访问
        row = valid_tfidf[index]
        sum_list = calculate_distance(train_tfidf, row, False)      # 这个
test和测试集的距离集合
        sort_index = np.argsort(sum_list)    # 返回排序之前最小的下标
        dict1 = {}
        i = 0
        while i < k:
            if train_labels[sort_index[i]] not in dict1:
                dict1[train_labels[sort_index[i]]] = 1
            else:
                dict1[train_labels[sort_index[i]]] += 1
            i += 1
        result[index] = max(dict1, key=lambda x: dict1[x])   # 找出字典中值最
大对应的键： lambda函数
    return result
```

- calculate_accuracy计算准确率函数：

```
def calculate_accuracy(valid_real, valid_predict):
    """
    :param valid_real: 真实的验证集标签
    :param valid_predict: 预测的验证集标签
    :return: 返回准确率
    """
    total = len(valid_real)
    count = 0
    for i in range(total):
        if valid_real[i] == valid_predict[i]:
            count += 1
    return count/total
```

- main函数：

```
def main():
    train_lines, train_labels = read_file_csv('train_set.csv')
    valid_lines, valid_labels = read_file_csv('validation_set.csv')
    test_lines = read_file_csv('test_set.csv')
    word_list = count_words(train_lines, valid_lines, test_lines)
    train_tfidf, valid_tfidf, test_tfidf = tf_idf(train_lines, valid_lines,
test_lines, word_list)
    ''' 调参：k
    k = 3
    while k < 20:
        valid_predict = KNN_predict(train_tfidf, valid_tfidf, k,
train_labels)
        accuracy = calculate_accuracy(valid_labels, valid_predict)
        print('k = '+str(k)+', accuracy = '+str(accuracy));
        k += 1
    '''
    test_predict = KNN_predict(train_tfidf, test_tfidf, 13, train_labels)
    test_output = pd.DataFrame({'Words (split by space)': test_lines,
'label': test_predict})
    test_output.to_csv('18340159_tangry_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号
```

4. 创新点

在表示 `tf-idf` 矩阵和计算距离的过程中，引入了 `numpy` 矩阵，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。大大加速效果！

5. 实验结果和分析

- **结果展示和分析**
  - 通过遍历不同的k值，循环调用KNN分类算法，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。经过测试，取k=3到20，依次输出精确度和对应k值。
  - 当距离使用**曼哈顿距离**衡量时，精确度随k值的变化如下：

```
k = 3, accuracy = 0.37942122186495175
k = 4, accuracy = 0.40192926045016075
k = 5, accuracy = 0.40514469453376206
k = 6, accuracy = 0.40192926045016075
k = 7, accuracy = 0.3987138263665595
k = 8, accuracy = 0.3858520900321543
k = 9, accuracy = 0.36977491961414793
k = 10, accuracy = 0.38263665594855306
k = 11, accuracy = 0.3729903536977492
k = 12, accuracy = 0.3858520900321543
k = 13, accuracy = 0.4115755627009646
k = 14, accuracy = 0.3987138263665595
k = 15, accuracy = 0.3890675241157556
k = 16, accuracy = 0.3858520900321543
k = 17, accuracy = 0.3762057877813505
k = 18, accuracy = 0.38263665594855306
k = 19, accuracy = 0.36977491961414793
```

- 当距离使用**欧氏距离**衡量时，精确度随k值的变化如下：

```
k = 3, accuracy = 0.2379421221864952
k = 4, accuracy = 0.21864951768488747
k = 5, accuracy = 0.18971061093247588
k = 6, accuracy = 0.17684887459807075
k = 7, accuracy = 0.17684887459807075
k = 8, accuracy = 0.17041800643086816
k = 9, accuracy = 0.17041800643086816
k = 10, accuracy = 0.17041800643086816
k = 11, accuracy = 0.1832797427652733
k = 12, accuracy = 0.19614147909967847
k = 13, accuracy = 0.21221864951768488
k = 14, accuracy = 0.2347266881028939
k = 15, accuracy = 0.24437299035369775
k = 16, accuracy = 0.24115755627009647
k = 17, accuracy = 0.24115755627009647
k = 18, accuracy = 0.2379421221864952
k = 19, accuracy = 0.2540192926045016
k = 20, accuracy = 0.26366559485530544
k = 21, accuracy = 0.2733118971061093
k = 22, accuracy = 0.2765273311897106
k = 23, accuracy = 0.2733118971061093
k = 24, accuracy = 0.2508038585209003
k = 25, accuracy = 0.26366559485530544
k = 26, accuracy = 0.26688102893890675
```

- 可以看到准确度随着k值的变化没有一个很明显的变化趋势（可能是由于训练及验证样本数量较少），总体在摆动。
- 其中，距离取曼哈顿距离的话，在k=13的时候效果最好，达到了41.1%的准确率。

- **模型性能展示和分析**

以下比较基于TF-IDF方法：

	曼哈顿距离	欧氏距离	精确度	k值
初始	0	1	23.79%	3

	曼哈顿距离	欧氏距离	精确度	k值
优化1	0	1	27.65%	22
优化2	1	0	41.15%	13
最优效果	1	0	41.15%	13

- 可以看到，当P=1也就是取曼哈顿距离的时候，精确度最高，达到了41.15%。而欧氏距离在这里的表现并不好。

Task 3：KNN回归问题

1. 算法原理

- 算法原理和task 2 KNN分类问题基本一样，回归是预测连续值的问题，即本实验中预测各个例子的label的概率。
- 具体步骤： 通过将各个文本转化成向量形式，找到最近的k个向量，并读出他们对应的各个情感的比例。然后将距离的倒数作为权重，计算目标文本和临近k个向量的每个情感在距离的加权求和后的结果（以happy为例）

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

- 在回归问题中，相关系数是研究变量之间线性相关程度的量。公式如下：

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

- 将预测验证集得到的概率值代入X，将真实的概率值代入Y，然后分别计算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。

2. 伪代码

- 在task 2中，已经呈现了许多函数的伪代码，此处指写出不同的部分

```
Function KNN_predict(train_tfidf, valid_tfidf, k, train_labels)
    label_num := train_labels的大小
result := 初始化为取值为0的二维列表
    for index:=0 to 验证集大小:
        row := valid_tfidf[index]
        sum_list := call calculate_distance(train_tfidf, row, True) /*测量距离*/
        sort_index := np.argsort(sum_list)
        dict1初始化为空
        i := 0
        pro_sum := 0
        while i<k do
            for j:=0 to label:
                if sum_list[sort_index]=0 then
                    sum_list[sort_index] := 0.01
```

```
                result[j][index]+= train_labels[j][sort_index[i]] /
float(sum_list[sort_index[i]])
                更新pro_sum
                end if
            i += 1
            end for
        end
        for i:=0 to label_num:
            result[i][index] /= pro_sum /*归一化*/
        end for
    end for
    return result
```

```
Function calculate_cor(valid_real, valid_predict)
    valid_real := valid_real的np矩阵形式
    valid_predict := valid_predict的np矩阵形式
    label_num := valid_predict的label数
    correlation := np.corrcoef(valid_real, valid_predict)
    cor_sum := 0
    for i:=0 to label_num:
        cor_sum += correlation[i][label_num+i]
    end for
    cor := 0
    cor := cor_sum / label_num
    return cor
```

3. 代码展示

除了一下几个板块，其他函数与KNN分类问题的代码相同。

- KNN_predict函数，添加了归一化处理：

```
def KNN_predict(train_tfidf, valid_tfidf, k, train_labels):
    """
    :param train_tfidf: 训练集
    :param valid_tfidf: 验证集
    :param k: k值
    :param train_labels: 训练集的情感集合
    :return: 返回预测的情感集合
    """
    label_num = len(train_labels)
    result = [[0.0 for j in range(valid_tfidf.shape[0])]for i in
range(label_num)]
    for index in range(0, valid_tfidf.shape[0]):      # ！！遍历验证集/测试集的
test
        row = valid_tfidf[index]
        sum_list = calculate_distance(train_tfidf, row, True)     # 这个test
和测试集的距离集合
        sort_index = np.argsort(sum_list)    # 返回排序之前最小的下标
        i = 0
        pro_sum = 0
        while i < k:
            for j in range(label_num):
                if sum_list[sort_index[i]] == 0:
                    sum_list[sort_index[i]] = 0.01
```

```
                result[j][index] += train_labels[j][sort_index[i]] /
float(sum_list[sort_index[i]])
                pro_sum += train_labels[j][sort_index[i]] /
float(sum_list[sort_index[i]])
            i += 1
        for i in range(label_num):
            result[i][index] /= pro_sum     # 归一化
    return result
```

- 计算相关系数，调用了numpy库中的corrcoef函数：

```
def calculate_cor(valid_real, valid_predict):
    """
    :param valid_real: 真实的验证集标签
    :param valid_predict: 预测的验证集标签
    :return: 返回相关系数
    """
    valid_real = np.array(valid_real)        # 转成np矩阵
    valid_predict = np.array(valid_predict)
    label_num = valid_predict.shape[0]
    test_num = valid_predict.shape[1]
    correlation = np.corrcoef(valid_real, valid_predict)
    cor_sum = 0
    for i in range(label_num):
        cor_sum += correlation[i][label_num+i]
    cor = 0
    cor = cor_sum / label_num
    return cor
```

- main函数：

```
def main():
    train_lines, train_labels = read_file_csv('train_set1.csv')
    valid_lines, valid_labels = read_file_csv('validation_set1.csv')
    test_lines = read_file_csv('regression_simple_test.csv')
    word_list = count_words(train_lines, valid_lines, test_lines)
    train_tfidf, valid_tfidf, test_tfidf = tf_idf(train_lines, valid_lines,
test_lines, word_list)
    ''' 调参：k
    k = 3
    while k < 20:

        valid_predict = KNN_predict(train_tfidf, valid_tfidf, k,
train_labels)
        cor = calculate_cor(valid_labels, valid_predict)
        print('k = ' + str(k) + ', Correlation coefficient = ' + str(cor))
        k += 1
    '''

    test_predict = KNN_predict(train_tfidf, test_tfidf, 14, train_labels)
    test_output = pd.DataFrame({'Words (split by space)': test_lines,
'anger': test_predict[0], 'disgust': test_predict[1],
                                'fear': test_predict[2], 'joy':
test_predict[3], 'sad': test_predict[4], 'surprise': test_predict[5]})
    test_output.to_csv('18340159_tangry_KNN_regression_sample.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号
```

4. 实验结果与分析

- **结果展示和分析**
  - 通过遍历不同的k值，循环调用KNN回归算法，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回归系数。经过测试，取k=3到30，依次输出回归系数和对应k值。
  - 当距离是曼哈顿距离的时候，结果如下：

```
k = 3, Correlation coefficient = 0.31635738697656807
k = 4, Correlation coefficient = 0.32525132555084196
k = 5, Correlation coefficient = 0.3227160899046577
k = 6, Correlation coefficient = 0.32522254267169204
k = 7, Correlation coefficient = 0.31211085366060115
k = 8, Correlation coefficient = 0.316372564257192
k = 9, Correlation coefficient = 0.30785183237412894
k = 10, Correlation coefficient = 0.31469633558551763
k = 11, Correlation coefficient = 0.32209775996910595
k = 12, Correlation coefficient = 0.32451177938791037
k = 13, Correlation coefficient = 0.3269034632830868
k = 14, Correlation coefficient = 0.33245083187909436
k = 15, Correlation coefficient = 0.32580554764405917
k = 16, Correlation coefficient = 0.326640413606699
k = 17, Correlation coefficient = 0.3264130073165443
k = 18, Correlation coefficient = 0.32156733955725253
k = 19, Correlation coefficient = 0.32148629511863946
k = 20, Correlation coefficient = 0.32196083321168806
k = 21, Correlation coefficient = 0.31876837599219315
k = 22, Correlation coefficient = 0.3189071208515279
k = 23, Correlation coefficient = 0.31062181970795194
k = 24, Correlation coefficient = 0.3055279928264838
k = 25, Correlation coefficient = 0.3063278964121714
k = 26, Correlation coefficient = 0.3037533851319792
k = 27, Correlation coefficient = 0.3037702987424553
```

  - 当距离是欧式距离的时候，结果如下：

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

  - 经过比较分析，可以看到准确度随着k值的变化没有一个很明显的变化趋势（可能是由于训练及验证样本数量较少），总体在摆动。
  - 并且，在取曼哈顿距离的时候，k=14的时候，相关系数取到最大值为33.24%。
- **模型性能展示和分析**

以下比较基于TF-IDF方法：

	曼哈顿距离	欧氏距离	相关系数	k值
初始	1	0	31.63%	3
优化1	1	0	33.24%	14
优化2	0	1	26.02%	17
最优效果	1	0	33.24%	14

- 可以看到，当P=1也就是取曼哈顿距离的时候，相关系数最高，达到了33.24%。而欧氏距离在这里的表现并不好。结果与分类问题类似。