KNN 分类和回归

- **算法原理**

（1）SemEval

首先对 semeval.txt 的数据进行读取，将数据存储在 str_list 列表中，构造无重复词语列表 word_list，通过两层循环将每个句子中没有重复出现的词语加入到列表中。接着构造 tf 表，对每一行的句子的每一个词语，得到词语在 word_list 出现的位置，在 tf 相应位置中加上词语在当前句子中出现的频率，然后建立 df 表，通过双层循环，得出 word_list 中每个词语在出现在句子的次数，应用 idf 公式构造 df 表，最后根据 tf_idf 公式通过双层循环，将 tf 表与 idf 表对应位置进行相乘得到对应的 tf_idf 表。

（2）KNN 分类

1. 首先分别读取训练集和验证集的 csv 文件数据，分别将数据传到列表 sample_list、validation_list，通过两层循环建立无重复词语列表 word_list，分别构建训练集和验证集的 one_hot 矩阵，对每一行的句子的每一个词语，找出词语在 word_list 出现的下标，在 one_hot 矩阵相应的对应位置置为 1。

2. 采取不同的度量方式，分别有汉明距离、杰卡德距离、余弦相似度、曼哈顿距离、欧式距离，比较不同度量方式的准确率。对于验证集的每一行句子，采用字典结构存储当前句子与训练集所有句子的距离或余弦相似度，字典的 key 为训练集句子的下标，value 为验证集当前句子与训练集句子的距离或者余弦相似度，利用 operator 模块对字典进行排序（距离按照从小到大排序，余弦相似度从大到小排序），选出前 k 个作为最终选取的参考，此时结构为列表，每个元素为一个元组，每个元组包含两个元素，第一个为训练集句子下标，第二个为距离或相似度。

3. 对 k 个列表，以距离为参考，采用字典统计各种情感的比重，key 为情感 label，value 为比重，考虑到距离越近，说明越接近测试数据，因此应该给予更多的权重，所以将距离的倒数的 3 次方（经过测试，3 次方准确率最高）作为该距离的比重添加到对应的情感 label，最后选出情感比重最大的 label 作为最终的结果。

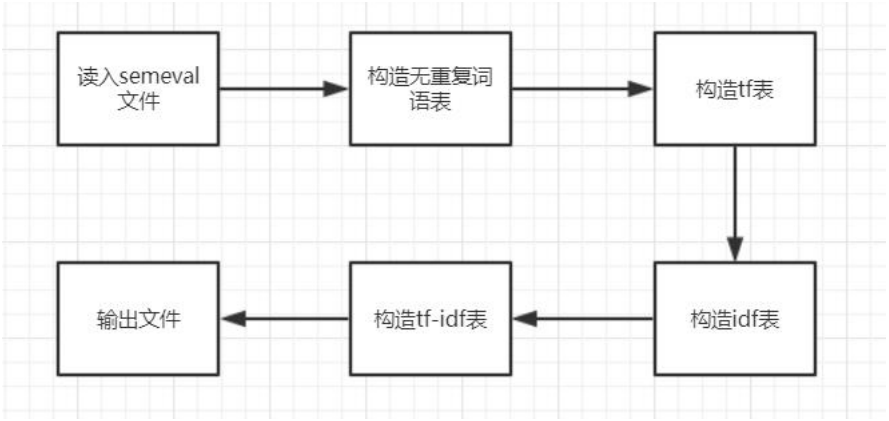
（3）KNN 回归

1. KNN 回归与 KNN 分类的 1、2 步相同，因此不再重复叙述。

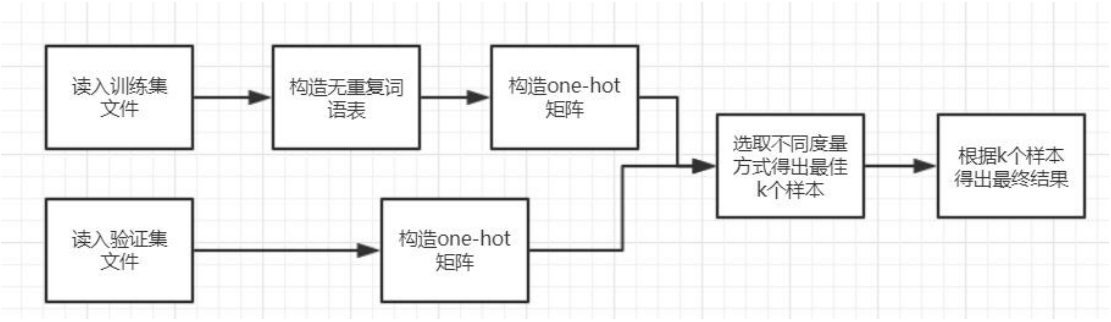
2. KNN 回归第 3 步跟 KNN 分类地 3 步不同，以余弦相似度为例，首先将 k 个相似度乘以 15+1 的 5 次方（经测试相关系数最高）作为权重，相加起来得到总权值，根据当前相似度的权值/总权值作为比重，以 joy 情感为例，则预测 joy 的概率 $p = p_1 * n_1 + ... + p_k * n_k$，依次类推，得到最终的预测值，计算相关系数，比较准确率。

- **伪代码或流程图**

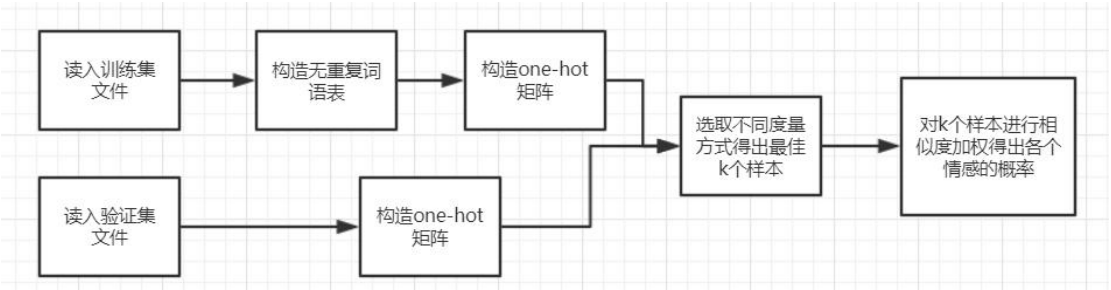
（1）Semeval



（2）KNN 分类



（3）KNN 回归



- 关键代码

（1）Semeval

1. 构造无重复词语列表

```
word_list = []  #无重复词语列表
length = len(str_list)
#构造无重复词语列表
for i in range(length):
    tmp_list = str_list[i].split()[8:]  #取出每一行后面的句子，前面的情感不是本程序研究的内容
    length1 = len(tmp_list)
    for j in range(length1):
        if tmp_list[j] not in word_list:   #如果当前词语没有出现过，则添加该词语到列表中
            word_list.append(tmp_list[j])
    str_list[i] = tmp_list   #将句子重新赋给句子列表
```

2. 构造 tf 表

```
for i in range(length):
    size = len(str_list[i])  #句子词语的总个数
    for j in range(size):
        # for k in range(length2):
        #     if str_list[i][j] == word_list[k]:
        k = word_list.index(str_list[i][j])   #在无重复词语列表查找句子当前的词语的下标
        tf[i][k] += 1/size  #在该词语的下标上加上权值1/size
```

3. 构造 idf 表

```
#构造idf表
for i in range(length2):
    num = 0  #用于记录词语出现在多少个句子中
    for j in range(length):
        if word_list[i] in str_list[j]:  #如果词语在当前句子中出现，则记录数加1
            num+=1

    idf_value = math.log(length / (num+1) )  #应用计算公式
    idf.append(idf_value)
```

4. 构造 tf_idf 表

```
tf_idf = []  #tf_idf表
#构造tf_idf表
for i in range(length):
    tmp_list2 = []
    for j in range(length2):
        value = tf[i][j] * idf[j]   #根据公式，将tf每一行的列表与idf列表进行点乘
        tmp_list2.append(value) #添加value进临时列表
    tf_idf.append(tmp_list2)  #将临时列表添加到tf_idf列表
```

（2）KNN 分类

1. 构造 one_hot 矩阵

```python
def build_one_hot(sample_list,word_list):
    # 建立训练集的onehot矩阵
    size1 = len(sample_list)
    size3 = len(word_list)
    one_hot = []
    for i in range(size1):
        tmp_list = [0] * size3  # 初始化为0
        one_hot.append(tmp_list)
    # 建立onehot矩阵
    for i in range(size1):
        tmp_list = sample_list[i][0].split()
        size2 = len(tmp_list)
        for j in range(size2):
            # for k in range(size3):
            #     if tmp_list[j] == word_list[k]:
            #         one_hot[i][k] = 1
            #         break
            k = word_list.index(tmp_list[j])  # 查找句子中的词语对应在word_list的下标
            one_hot[i][k] = 1
    return one_hot
```

2. 计算曼哈顿距离

```python
#求曼哈顿距离
def get_manhattan_distance(one_hot,validation_one_hot,x,sample_list_size):
    distance = {}
    for i in range(sample_list_size):
        # dis_tmp = 0
        # for j in range(word_list_size):
        #     dis_tmp += abs(one_hot[i][j] - validation_one_hot[x][j])
        a1 = np.array(one_hot[i])   #将列表转换为numpy的array
        a2 = np.array(validation_one_hot[x])
        distance[i] = np.linalg.norm(a1 - a2, ord=1)   #应用numpy的函数计算
    return distance
```

3. 计算欧式距离

```python
#求欧氏距离
def get_euclidean_distance(one_hot, validation_one_hot, x, sample_list_size):

    distance = {}
    for i in range(sample_list_size):
        # dis_tmp = 0
        # for j in range(word_list_size):
        #     dis_tmp += math.pow((one_hot[i][j] - validation_one_hot[x][j]), 2)
        # dis_tmp = math.pow(dis_tmp, 0.5)
        a1 = np.array(one_hot[i])     #将列表转换为numpy的array
        a2 = np.array(validation_one_hot[x])
        dis_tmp = np.linalg.norm(a1 - a2)  #应用numpy函数计算二范数即欧式距离
        distance[i] = dis_tmp
    return distance
```

4. 计算余弦相似度

```python
#余弦相似度
def get_cosine_similarity(one_hot, validation_one_hot, x, sample_list_size):
    cosine_similarity = {}
    for i in range(sample_list_size):

        if one_hot[i] == validation_one_hot[x]:
            cosine_similarity[i] = 1
        else:
            a1 = np.array(one_hot[i])
            a2 = np.array(validation_one_hot[x])
            cosine_similarity[i] = np.dot(a1, a2) / (np.linalg.norm(a1) * (np.linalg.norm(a2)))    #应用求两向量余弦公式
    return cosine_similarity
```

5. 计算汉明距离

```python
def get_hanming_distance(one_hot, validation_one_hot, x, sample_list_size):
    distance = {}
    for i in range(sample_list_size):
        a1 = np.array(one_hot[i])  #转换为array
        a2 = np.array(validation_one_hot[x])
        smstr = np.nonzero(a1-a2)
        distance[i] = np.shape(smstr[0])[0]
    return distance
```

6. 计算杰卡德距离

```python
def get_jake_distance(one_hot, validation_one_hot, x, sample_list_size):
    distance = {}
    for i in range(sample_list_size):
        a1 = np.array(one_hot[i])    #转换为array
        a2 = np.array(validation_one_hot[x])
        matv = np.array([a1,a2])
        distance[i] = dist.pdist(matv,'jaccard')
    return distance
```

7. 根据 k 的推荐值计算出最终的情感结果

```
    #遍历，统计各个label出现的次数
    for i in range(size):
        index = sort_distance[i][0]
        if sort_distance[i][1] == 0:  #如果距离为0，说明两个句子完全吻合，赋予极大权值
            weight = 10000
        else:
            weight = 1 / (sort_distance[i][1])   #赋予距离倒数，距离越小，倒数越大
        emotion = sample_list[index][1]
        if emotion in fre.keys():
            # 为了让距离更小的有更大的权值，对当前的权值进行幂次运算，增大权重
            fre[emotion] += math.pow(weight,weight_coefficient)
        else:
            fre[emotion] = math.pow(weight,weight_coefficient)

max_value = max(fre.values())  #找出权重最大的label
for key,value in fre.items():
    if value == max_value:   #返回权重最大的label
        return key
```

（3）KNN 回归

1. 由于 1 跟 6 与 KNN 分类重复，不再叙述。
2. 根据 k 个推荐值对各个情感的概率进行加权归一。

```
    sum1 = 0
    for i in range(k_value):
        sum1 += math.pow((sorted_data[i][1] * 15 + 1), weight_coefficient) #求出总权值
        # sum1 += math.pow(math.e,(sorted_data[i][1]*15+1))
    for i in range(k_value):
        index = sorted_data[i][0]  #得到下标
        min = math.pow((sorted_data[i][1] * 15 + 1), weight_coefficient)
        # min = math.pow(math.e,(sorted_data[i][1]*15+1))
        weight = min / sum1  #权值占比
        anger += float(sample_list[index][1]) * weight  #依次乘以权值占比
        disgust += float(sample_list[index][2]) * weight
        fear += float(sample_list[index][3]) * weight
        joy += float(sample_list[index][4]) * weight
        sad += float(sample_list[index][5]) * weight
        surprise += float(sample_list[index][6]) * weight
```

- **创新点&优化**

（1）SemeEval

1. 最初开始构造 tf 表、idf 表和 tf_idf 表是通过暴力遍历方法构造，因此程序运行的效率很低，测试运行时间有 8.59s，显然这样的算法效率很低。

```
run time:   8.594781150247806  s
```

虽然后来做了一些小改动，程序运行快了 2s，为了详细分析程序各部分的运行时间，我测试了主要部分的运行时间，由图可见，构造 tf 表的时间最多，应该着重优化。

```
初始化:  0.1250527198137768  s
初始化tf表:  0.39387487327421666  s
得出tf表:  3.47695525848284  s
得出idf表:  0.8410850181333505  s
得出tf_idf表:  0.956265870221471  s
写出文件:  0.4602231128949743  s
总运行时间:  6.253740416510127  s
```

2. 为了进一步优化时间，提高程序运行效率，一开始打算用 python 的 numpy 中 array 去替代一部分列表，得出的效果不明显，程序反而变得更慢，构造 tf 表时间达到 11s，说明该方法不可靠，因此后来我没用 numpy 的 array。

```
初始化:  0.12968942035579503  s
初始化tf表:  6.683192149326e-05  s
得出tf表:  11.344967122092859  s
得出idf表:  2.0760233873967078  s
得出tf_idf表:  1.9362144058661066  s
写出文件:  0.6171584350758099  s
总运行时间:  16.10441298351668  s
```

3. 由于初始化 tf 表用了 0.4s，我觉得不合理，因此先从这部分下手优化，一开始我是两层遍历加 0，这样子循环次数达到 length*length2，直接对每一行进行优化，循环次数减少到 length，时间减少到 0.03s 左右。

```
tf = []
length2 = len(word_list)
for i in range(length):
    tmp_list1 = [0]*length2
    tf.append(tmp_list1)

# length2 = len(word_list)
```

4. 接下来着重优化最花时间的构造 tf 表，由代码可见，我用了 3 重循环，时间开销大，认真分析，发现找出当前句子的每个词语在 word_list（无重复词语列表）的 index 可以通过列表的 index 函数得出，从而化三重循环为二重循环，时间由 3.4s 降低为 0.13s。

```
for i in range(length):
    size = len(str_list[i])
    for j in range(size):
        # for k in range(length2):
        #     if str_list[i][j] == word_list[k]:
        k = word_list.index(str_list[i][j])
        tf[i][k] += 1/size
```

经过一系列优化，时间由 6.2s 降到 2.5s，程序运行效率有了约 2.5 倍提高。

```
初始化:  0.1296263642490754  s
初始化tf表:  0.029018646861259573  s
得出tf表:  0.13314617878105373  s
得出idf表:  0.8559534215489524  s
得出tf_idf表:  0.9531051356744642  s
写出文件:  0.45611842465456043  s
总运行时间:  2.557240030433067  s
```

（2）KNN classification & regression

1. 一开始是用简单的循环方式编写函数，计算欧式距离时，用了两重循环，循环次数为 sample_list_size*word_list_size，由于两个数值比较大，并且验证集的数据量有 300，因此运算有 300*500*2000，对于一个 k 值来说，运行程序一次花费时间为 170s，效率太低，为了优化效率，使用了 python 的 numpy 模块的函数取代直接暴力遍历的方法来求欧式距离

```
#求欧氏距离
def get_euclidean_distance(one_hot,validation_one_hot,x,sample_list_size):

    distance = {}
    for i in range(sample_list_size):
        # dis_tmp = 0
        # for j in range(word_list_size):
        #     dis_tmp += math.pow((one_hot[i][j] - validation_one_hot[x][j]), 2)
        # dis_tmp = math.pow(dis_tmp, 0.5)
        a1 = np.array(one_hot[i])     #将列表转换为numpy的array
        a2 = np.array(validation_one_hot[x])
        dis_tmp = np.linalg.norm(a1 - a2)  #应用numpy函数计算二范数即欧式距离
        distance[i] = dis_tmp
    return distance
```

通过对距离方式的优化，程序运行时间缩短到 50s。

2. 同样的思路，对求余弦相似度进行优化，提高了效率，同时减少代码量。

```
#余弦相似度
def get_cosine_similarity(one_hot, validation_one_hot, x, sample_list_size):
    cosine_similarity = {}
    for i in range(sample_list_size):
        # vector_product = 0
        # mode_tmp1 = 0
        # mode_tmp2 =0
        # for j in range(word_list_size):
        #     vector_product += one_hot[i][j] * validation_one_hot[x][j]
        #     mode_tmp1 += one_hot[i][j] * one_hot[i][j]
        #     mode_tmp2 += validation_one_hot[x][j] * validation_one_hot[x][j]
        # mode_tmp1 = math.pow(mode_tmp1, 0.5)
        # mode_tmp2 = math.pow(mode_tmp2, 0.5)
        # mode = mode_tmp1 * mode_tmp2
        # if mode == 0:
        #     cosine_similarity[i] = 0
        # else:
        #     cosine_similarity[i] = vector_product / (mode_tmp1 * mode_tmp2)
        if one_hot[i] == validation_one_hot[x]:
            cosine_similarity[i] = 1
        else:
            a1 = np.array(one_hot[i])
            a2 = np.array(validation_one_hot[x])
            cosine_similarity[i] = np.dot(a1, a2) / (np.linalg.norm(a1) * (np.linalg.norm(a2)))    #应用求两向量余弦公式
    return cosine_similarity
```

3. 求距离时，用函数对距离进行排序，避免暴力遍历。

```
print(distance)
sorted_distance = sorted(distance.items(), key=operator.itemgetter(1))
print(sorted_distance)
```

```
min = 100000000
index = 0
size4 = len(distance)
for i in range(size4):
    if min > distance[i]:
        min = distance[i]
        index = i
```

4. 根据分析，距离越近，权重应该要越大，取 k 个最近距离时，如果单纯取众数，会导致结果准确率不高，通过这样的分析，应该给距离远近加一个权重，引入幂次运算，对权重的倒数进行幂次方，从而提高权重，经测试，分类任务最高准确率由 38%上升到 44%，回归任务的相关系数由 0.22 上升到 0.29。

```
------
k w m :  13 2 0    correct rate:  0.4437299035369775
run time:   48.789685010442014  s
```

5. 采用多种度量方式，包括曼哈顿距离，欧式距离，余弦相似度，汉明距离，杰卡德距离。

汉明距离是两个等长 01 串 s1 与 s2 之间，一个变为另外一个所需要作的最小替换次数，适用于 one-hot 矩阵，比如 1111 替换为 1001 的汉明距离为 2，在 one-hot 中，汉明距离效果等同于曼哈顿距离，为 knn_classification 最优的度量方式。

```
def get_hanming_distance(one_hot, validation_one_hot, x, sample_list_size):
    distance = {}
    for i in range(sample_list_size):
        a1 = np.array(one_hot[i])  #转换为array
        a2 = np.array(validation_one_hot[x])
        smstr = np.nonzero(a1-a2)
        distance[i] = np.shape(smstr[0])[0]
    return distance
```

杰卡德距离是用两个集合中不同元素占所有元素的比例来衡量两个集合的区分度，也是一种比较好的度量方式；对于 knn_regression，效果最好的是余弦相似度。

```
def get_jake_distance(one_hot, validation_one_hot, x, sample_list_size):
    distance = {}
    for i in range(sample_list_size):
        a1 = np.array(one_hot[i])   #转换为array
        a2 = np.array(validation_one_hot[x])
        matv = np.array([a1, a2])
        distance[i] = dist.pdist(matv, 'jaccard')
    return distance
```

6. 对于回归任务，根据分析，余弦相似度效果最好，并且由于数值在 0-1，所以不能单纯地进行幂次方运算增大权重，而我先将其乘上一个整数，再进行幂次运算，经测试，整数为 15 时，效果最好，相关系数由 0.29 上升到 0.38。

	anger	disgust	fear	joy	sad	surprise
r	0.38374593	0.330105377	0.364219426	0.417663536	0.418197457	0.377772793
average	0.381950753					
evaluation	低度相关 666					

• 实验结果及分析

• 1. 实验结果展示

（1）Semeval

以下是部分 tf_idf 表的内容：

1.0724244197979087 1.0048469017798813 0.8217448536685299 0.5393122335394619 1.0048469017798813 0.7604573903143101
1.435349834556877 1.6086366296968633 0.5829757884376643 1.6086366296968633
0.7604573903143101 0.8893223716865571 1.0724244197979087 0.8413753596112603 0.5535052015961798 0.919709297818883
2.1448488395958174 1.190781879285995 1.9137997794091692
0.9568998897045846 0.672775207664847 1.0724244197979087 0.919709297818883 0.8041847677255587 1.0724244197979087
0.8893223716865571 1.0724244197979087 0.8893223716865571 0.4683675976351813 0.7604573903143101 1.0724244197979087
0.8302578023942697 1.6086366296968633 1.5072703526698221 1.6086366296968633
0.919709297818883 1.0724244197979087 0.7366072497075313 0.400050980105843 0.8636305917153475 0.9568998897045846
1.9137997794091692 2.1448488395958174 1.6434897073370598

（2）KNN 分类

部分输出结果：

28	joy	39	joy	69	joy
29	sad	40	fear	70	joy
30	fear	41	fear	71	anger
31	joy	42	fear	72	joy
32	sad	43	joy	73	fear
33	sad	44	joy	74	joy
34	joy	45	joy	75	joy
35	fear	46	fear	76	joy
36	joy	47	joy	77	fear
37	joy	48	joy	78	fear
38	joy	49	joy	79	fear

（3）KNN 回归

部分输出结果：

0.155034	0.074298	0.307332	0.058506	0.279511	0.125336
0.081944	0.080264	0.053186	0.5224	0.100835	0.161459
0.025778	0.063865	0.238722	0.174349	0.073252	0.424096
0.087052	0.055486	0.232488	0.190653	0.172496	0.261816
0.033748	0.026038	0.182402	0.430604	0.085537	0.241665
0.026594	0.043694	0.245651	0.030842	0.550575	0.102448
0.087294	0.111106	0.107699	0.087472	0.371365	0.235047
0.016496	0.021909	0.044743	0.020841	0.665369	0.230644
0.093889	0.071633	0.210652	0.283739	0.175535	0.164542
0.002	0.001546	0.005087	0.722282	0.011799	0.257286
0.044521	0.017207	0.041872	0.479684	0.124706	0.291986
0.011065	0.007547	0.101067	0.519923	0.184247	0.176169
0.000178	0.000108	0.361794	0.042272	0.162414	0.433197

经验证，六种情感概率相加约等于 1。

1.0000999524256402
0.9999877419029946
0.9999954268528852
1.0
0.9999816065141034
0.9999696384222487
0.9999862686509132
1.0000028087247577
0.9999696384222487

2. 评测指标展示

（1）knn 回归

1. 先确定权重系数 m（用于幂次运算的 m 次方）的最佳数值。

```
weight = sort_distance[i][1] * 15 + 1 # 赋予距离倒数，距离越小，倒数越大
emotion = sample_list[index][1]
if emotion in fre.keys():
    fre[emotion] += math.pow(weight, weight_coefficient)  # 为了让距离更小的有
else:
    fre[emotion] = math.pow(weight, weight_coefficient)
```

m	1	2	3	4	5
准确率	0.4405144694	0.4437299035	0.4340836012	0.424437299	0.38906752411

由表可得最佳 m 值为 2。

2. 确定最佳度量方式。

在 m 值为 2 的情况下，测量不同距离的在不同的 k 值的最优准确率。

度量方式	曼哈顿距离	欧式距离	余弦相似度	汉明距离	杰卡德距离
准确率	0.4437299035	0.4405144694	0.4276527331	0.443729903	0.44372990353

因此可得最佳度量方式为曼哈顿距离或汉明距离。

3. 确定最佳 k 值。

选取曼哈顿距离方式、m 值为 2，得出不同 k 值的最佳准确率。

由于 k 值在 1-10 中，准确率过低，20 以上后准确下降，因此只考虑 11-20.

k	11	12	13	14	15
准确率	0.43086816	0.42443729	0.44372990	0.43408360	0.43086816
k	16	17	18	19	20
准确率	0.43086816	0.43408360	0.44051446	0.42443729	0.42765273

最佳方式为曼哈顿距离或汉明距离，k 值为 13 ，m 值为 2 ，最佳准确率为 0.44372990 。

（2）knn 分类

1. 首先确定权重系数 m（用于幂次运算的 m 次方）的最佳数值

```
for i in range(k_value):
    sum1 += math.pow((sorted_data[i][1] * 15 + 1), weight_coefficient) #求出总权值
    # sum1 += math.pow(math.e,(sorted_data[i][1]*15+1))
for i in range(k_value):
    index = sorted_data[i][0]  #得到下标
    min = math.pow((sorted_data[i][1] * 15 + 1), weight_coefficient)
    # min = math.pow(math.e,(sorted_data[i][1]*15+1))
    weight = min / sum1  #权值占比
    anger += float(sample_list[index][1]) * weight  #依次乘以权值占比
    disgust += float(sample_list[index][2]) * weight
    fear += float(sample_list[index][3]) * weight
    joy += float(sample_list[index][4]) * weight
    sad += float(sample_list[index][5]) * weight
    surprise += float(sample_list[index][6]) * weight
```

m	2	3	4	5	6
相关系数	0.349899342	0.370673285	0.38002856	0.381950753	0.379426121

由表得出最佳权重系数为 5。

2. 在选取 m 为 5，测量不同度量方式的不同 k 值的最佳相关系数。

曼哈顿距离

	anger	disgust	fear	joy	sad
r	0.297954729	0.241821947	0.255660048	0.336731758	0.355374615
average	0.299797699				
evaluation	极弱相关 加油哦小辣鸡				

欧式距离

	anger	disgust	fear	joy	sad
r	0.288332637	0.232788455	0.269508168	0.319140703	0.319446221
average	0.285694936				
evaluation	极弱相关 加油哦小辣鸡				

余弦相似度

	anger	disgust	fear	joy	sad
r	0.38374593	0.330105377	0.364219426	0.417663536	0.418197457
average	0.381950753				
evaluation	低度相关 666				

汉明距离

	anger	disgust	fear	joy	sad
r	0.297954729	0.241821947	0.255660048	0.336731758	0.355374615
average	0.299797699				
evaluation	极弱相关 加油哦小辣鸡				

杰卡德距离

	anger	disgust	fear	joy	sad
r	0.305525836	0.223911257	0.309841325	0.35310201	0.347173742
average	0.312102232				
evaluation	低度相关 666				

根据各种度量方式比较得到最佳的度量方式为余弦相似度。

3. 最定最佳 k 值
选取余弦相似度、m 为 5，测量在不同 k 值下的相关系数（由于 k<10 相关系数低，k>20 相关系数下降，因此列表显示 11-20.）

k	11	12	13	14	15
相关系数	0.38219010	0.38304506	0.38418984	0.38301473	0.38333273
k	16	17	18	19	20
相关系数	0.38378301	0.38228894	0.38195075	0.38175585	0.38258957

由表可知，最佳方式为余弦相似度，k 值为 13，m 值为 5，最佳相关系数为 0.38418984。

• 拓展思考

1. IDF 的第二个计算公式中分母多了个 1 是为什么？
答：为了避免分母为 0，因为一个单词有可能都没有在任何文章中出现过，因此统计出现了该单词的文章总数为 0，加 1 有效避免了程序运行出错。

2. IDF 数值有什么含义？TF-IDF 数值有什么含义？
答：IDF 表示逆文本频率指数，是由总文档数除以包含该词语的文档数的商取以 10 为底的对数得到，用来衡量词语 m 出现在多少个文档，如果包含词语 m 的文档越少，则 IDF 的数值越大，表明该词语有很好的区分能力。TF-IDF 用于评估一个词语的重要程度，是由 tf（词语 m 在当前的文档出现的频率）与 IDF 相乘得到，如果 tf 数值大，即词语在当前文档出现的频率高，而 idf 数值大，即词语在出现在的文档数少，那么 tf-idf 数值就大，从而过滤常见词语，保留重要的词语。

3. 根据相似度加权中为什么取距离的倒数作为权重？
答：因为根据距离公式，两个句子相似度越高，距离应该是越小，权重应该要越大，因此需要将距离取倒数作为权重，同时为了让六种情感概率相加为 1，需要将其归一化。