**中山大学数据科学与计算机学院**

**计算机科学与技术专业-人工智能**

**本科生实验报告**

**（2018-2019学年秋季学期）**

课程名称：**Artificial Intelligence**

|  |  |  |  |
| --- | --- | --- | --- |
| 教学班级 | 计科2班 | 专业（方向） | 计算机科学与技术 |
| 学号 | 16337341 | 姓名 | 朱志儒 |

* **实验题目**

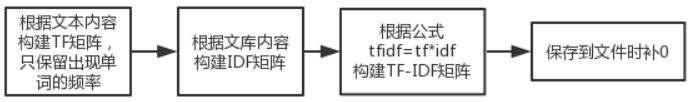
**TF-IDF实验**

* **实验内容**
* **算法原理**

由于semeval数据集中每一行语句中单词个数远小于整个数据集中的单词个数，所以构成的TF-IDF矩阵是一个稀疏矩阵，矩阵中存在许多0，因此在生成矩阵时，我选择只保留不为0的数据。

算法的整个过程：首先遍历整个文本来生成不重复词汇向量，接着根据词汇是否在每行文本中出现来生成ONE-HOT矩阵，再将ONE-HOT矩阵中的数据归一化生成TF矩阵，接着计算每个单词出现的文章总数，再生成IDF矩阵，最后根据公式：tfidfi,j = tfi,j \* idfi生成TF-IDF矩阵。

* **流程图**



* **关键代码截图**

for line in article:

#生成TF矩阵，只保留出现单词归一化后的频率

words = line.split(' ')

tmp = {}

for word in words:

if word not in tmp.keys():

tmp[word] = 1 / len(words)

else:

tmp[word] += 1 / len(words)

TF.append(tmp)

#去除每句文本中的重复单词

tmp = []

for element in words:

if element not in tmp:

tmp.append(element)

#记录单词在每句文本中出现的次数

for word in tmp:

if word not in IDF.keys():

IDF[word] = 1

else:

IDF[word] += 1

#生成IDF矩阵

for key in IDF.keys():

IDF[key] = math.log(len(article) / (IDF[key] + 1))

#根据公式生成TF-IDF矩阵

TF\_IDF = []

for i in range(len(TF)):

tmp = {}

for key in TF[i].keys():

tmp[key] = TF[i][key] \* IDF[key]

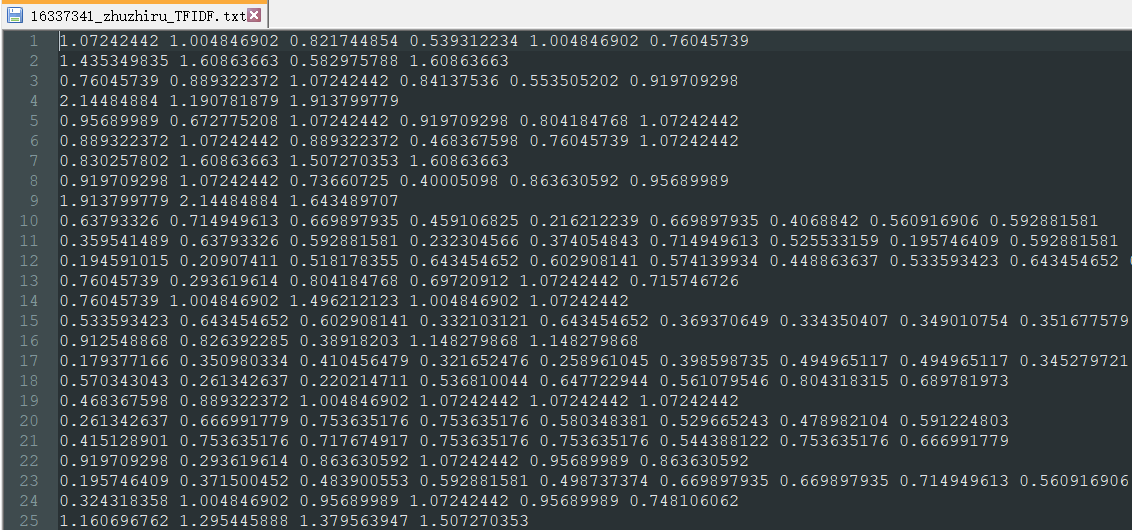
TF\_IDF.append(tmp)

* **创新点&优化**

TF和TF-IDF矩阵均为稀疏矩阵，在生成TF和TF-IDF矩阵时没有保存0，这样减少了占用的内存空间和运行时间。

* **实验结果**

**实验结果展示示例**



* **思考题**

**1. IDF的第二个计算公式中分母多了个1是为什么？**

分母多了个1是为了防止分母为0。

**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，IDF数值越小，说明该词在文库中越常见。TF-IDF数值越大，说明该词在文章中的重要性越高。

* **实验题目**

**k-NN处理分类问题**

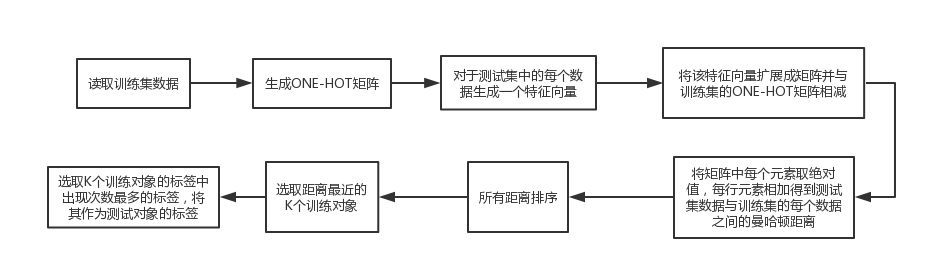
* **实验内容**
* **算法原理**

KNN算法，即K近邻分类算法，根据不同特征值之间的距离来进行分类的一种简单而又懒惰的算法。

KNN算法的步骤：对于每个测试集中的不带标签的对象，通过使用曼哈顿距离、欧式距离或夹角余弦来计算它与训练集中的每个对象的距离，接着选取距离最近的K个训练对象，然后在这K个训练对象的标签中选出出现次数最多的标签作为该测试集对象的标签。

在本次实验中，我选择使用曼哈顿距离计算测试集向量与训练集向量的距离。

* **流程图**



* **关键代码截图**

（1）对训练集数据的处理：

#读取训练集中的数据

train\_set = read\_csv('lab1\_data\\classification\_dataset\\train\_set.csv')

for i in range(len(train\_set)):

train\_set[i][0] = train\_set[i][0].split(' ')

#生成不重复词向量

words\_vector = []

for line in train\_set:

words\_vector += line[0]

words\_vector = list(set(words\_vector))

#生成训练集数据的ONE-HOT矩阵

xmatrix = np.zeros((len(train\_set), len(words\_vector)))

for i in range(len(train\_set)):

for word in train\_set[i][0]:

xmatrix[i][words\_vector.index(word)] += 1

xmatrix = np.matrix(xmatrix)

（2）实现KNN算法：

def classification\_KNN(test\_set, k):

labels = []

for line in test\_set:

words = line[0].split(' ')

test\_vector = np.zeros(len(words\_vector))

# 对于测试集中的每个数据生成一个特征向量

for word in words:

if word in words\_vector:

test\_vector[words\_vector.index(word)] += 1

# 将该特征向量扩展成矩阵并与训练集的ONE-HOT矩阵相减，再对得到的矩阵中每个元素取绝对值

test\_matrix = np.abs((np.tile(test\_vector, (len(train\_set), 1)) - xmatrix))

# 将矩阵中每行元素相加得到测试集数据与训练集的每个数据之间的曼哈顿距离

distance = np.sum(test\_matrix, axis=1).T.tolist()[0]

maps = {}

for i in range(len(distance)):

if distance[i] not in maps.keys():

maps[distance[i]] = [train\_set[i][1]]

else:

maps[distance[i]].append(train\_set[i][1])

# 将得到的所有距离排序

distance.sort()

# 选取距离最近的K个训练对象

count = 0

mood = []

for index in range(len(distance)):

i = distance[index]

if len(maps[i]) + count >= k:

mood += maps[i]

break

else:

mood += maps[i]

count += len(maps[i])

# 选取K个训练对象的标签中出现次数最多的标签

top = Counter(mood).most\_common(1)[0][0]

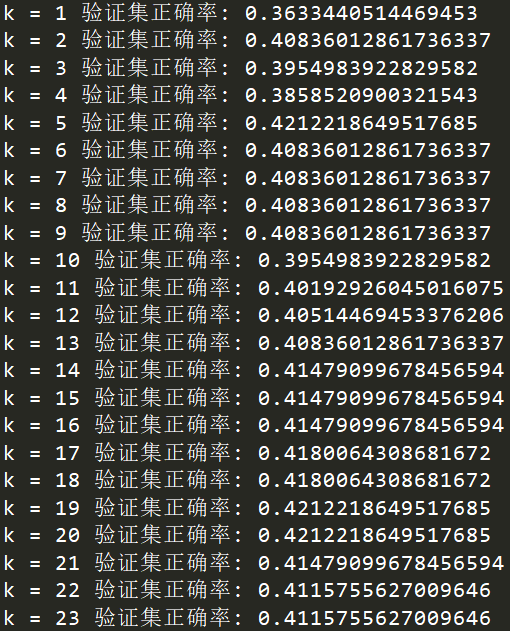
labels.append(top)

return labels

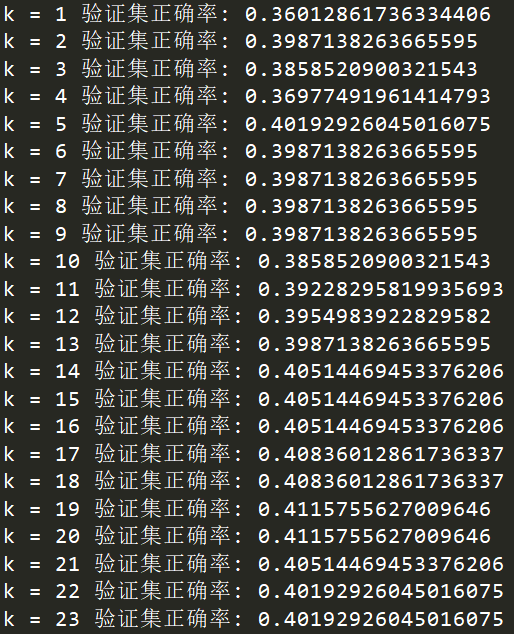
* **创新点&优化**

由于计算两个向量之间的距离有多种方式，本次实验我使用曼哈顿距离和欧式距离对比选择最优，试用不同的K值来选择最优的。

使用曼哈顿距离时：



使用欧式距离时：

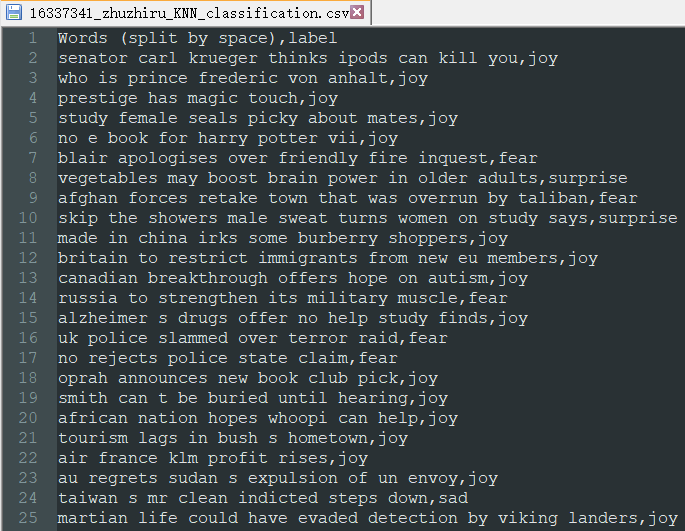


经过多次实验测试，最终发现使用曼哈顿距离，K=5时效果最佳。

* **实验结果及分析**

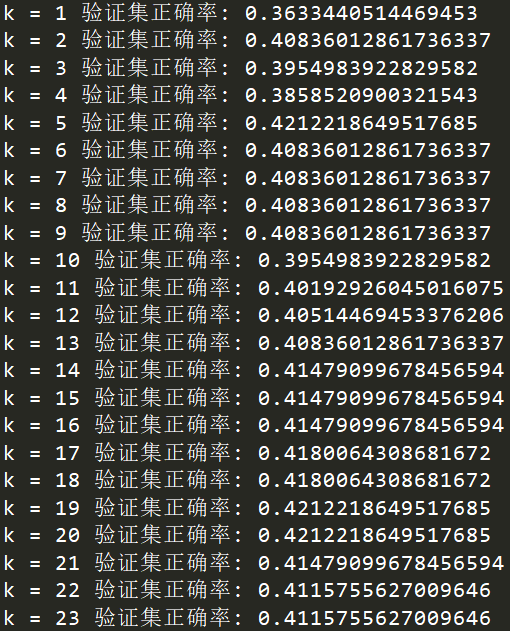
**1. 实验结果展示示例**

对测试集数据进行分类结果如图：



**2. 评测指标展示即分析**

选择使用曼哈顿距离，K=5时，验证集的准确率如图所示：



* **实验题目**

**k-NN处理回归问题**

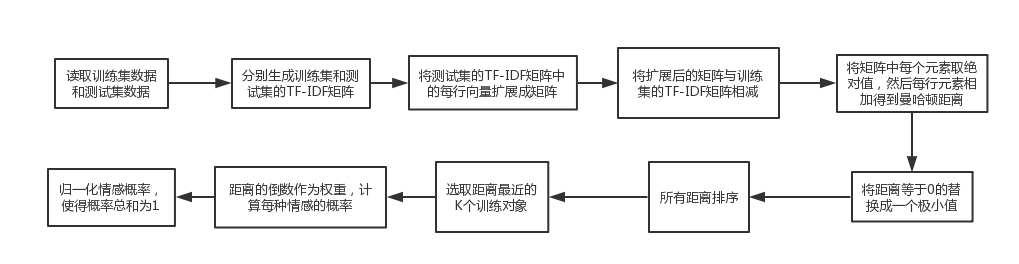
* **实验内容**
* **算法原理**

KNN算法，即K近邻分类算法，根据不同特征值之间的距离来进行分类的一种简单而又懒惰的算法。

KNN算法的步骤：对于每个测试集中的不带标签的对象，通过使用曼哈顿距离、欧式距离或夹角余弦来计算它与训练集中的每个对象的距离，接着选取距离最近的K个训练对象， 然后根据距离大小对这K个训练对象标签的概率加权求和得到测试对象的标签概率。

在本次实验中，我再次选择曼哈顿距离计算测试集向量与训练集向量之间的距离。与上一个实验不同的是，本次实验我为训练集数据和测试集数据分别生成了一个TF-IDF矩阵，使用TF-IDF矩阵中的向量计算距离。

* **流程图**



* **关键代码截图**

（1）生成TF-IDF矩阵的函数：

def sub\_TF\_IDF(test\_word\_vector, test\_set):

test\_TF = []

test\_IDF = [0 for i in range(len(test\_word\_vector))]

idf = {}

for line in test\_set:

words = line[0].split(' ')

# 生成TF矩阵

test\_vector = np.zeros(len(test\_word\_vector))

for word in words:

test\_vector[test\_word\_vector.index(word)] += 1 / len(words)

test\_TF.append(test\_vector)

# 记录单词在每句文本中出现的次数

unique = list(set(words))

for word in unique:

if word not in idf.keys():

idf[word] = 1

else:

idf[word] += 1

# 生成IDF矩阵

for key in idf.keys():

test\_IDF[test\_word\_vector.index(key)] = math.log(len(test\_set) / (idf[key] + 1))

# 根据公式生成TF-IDF矩阵

test\_TFIDF = test\_TF \* np.tile(test\_IDF, (len(test\_set), 1))

return test\_TFIDF

（2）KNN算法实现：

def regression\_KNN(test\_set, k):

labels = []

# 生成不重复词向量

test\_word\_vector = []

for row in test\_set:

test\_word\_vector += row[0].split(' ')

for row in train\_set:

test\_word\_vector += row[0].split(' ')

test\_word\_vector = list(set(test\_word\_vector))

# 生成训练集文本数据的TF-IDF矩阵

train\_TFIDF = sub\_TF\_IDF(test\_word\_vector, train\_set)

# 生成测试集文本数据的TF-IDF矩阵

test\_TFIDF = sub\_TF\_IDF(test\_word\_vector, test\_set)

for num in range(len(test\_set)):

test\_vector = test\_TFIDF[num]

# 将测试集中一行文本数据的TF-IDF向量扩展成矩阵，并与训练集的TF-IDF矩阵相减，再取绝对值

test\_matrix = np.abs(np.tile(test\_vector, (len(train\_set), 1)) - train\_TFIDF)

# 将矩阵中每行元素相加得到测试集数据与训练集的每个数据之间的曼哈顿距离

distance = np.sum(test\_matrix, axis=1).T.tolist()

# 将距离等于0的替换为一个极小值

for i in range(len(distance)):

if distance[i] == 0:

distance[i] = 0.0000000000001

maps = {}

for i in range(len(distance)):

if distance[i] not in maps.keys():

maps[distance[i]] = [i]

else:

maps[distance[i]].append(i)

# 将得到的所有距离排序

dist = sorted(distance)

count = 0

moods = []

# 选取距离最近的K个训练对象

for index in range(len(dist)):

dd = dist[index]

if len(maps[dd]) + count >= k:

moods += maps[dd]

break

else:

moods += maps[dd]

count += len(maps[dd])

# 计算测试对象的情感概率

sentiments = []

for i in range(1, 7):

sentiment = 0

for index in moods:

sentiment += float(train\_set[index][i]) / distance[index]

sentiments.append(sentiment)

# 归一化测试对象的情感概率，使得各个概率的总和为1

summ = sum(sentiments)

for i in range(len(sentiments)):

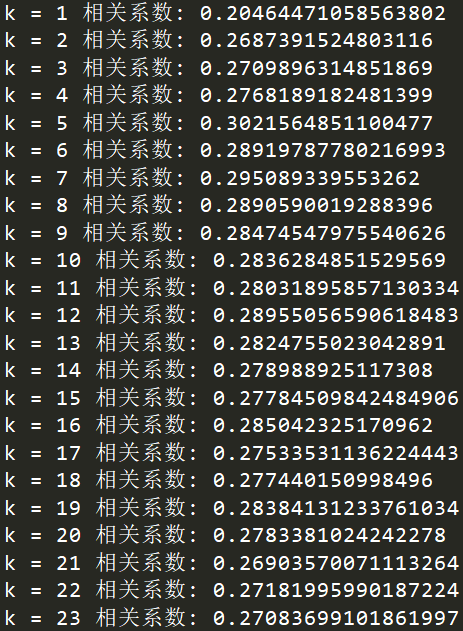
sentiments[i] = sentiments[i] / summ

labels.append(sentiments)

return labels

* **创新点&优化**

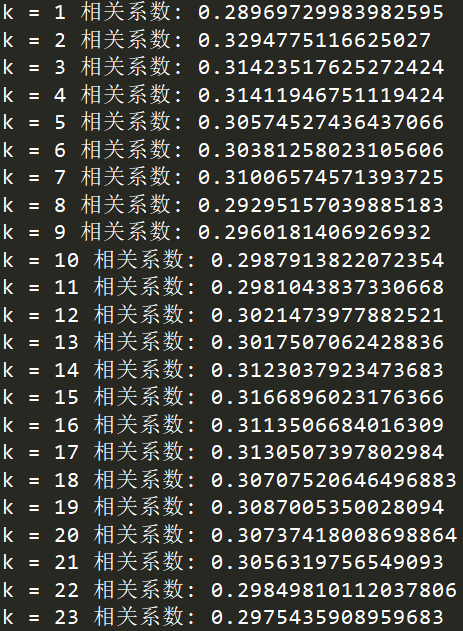
之前在处理训练集和测试集的数据时，像上一个实验一样，分别生成一个ONE-HOT矩阵，再使用曼哈顿距离计算向量之间的距离，选取不同的K值分别预测验证集，然而最终的平均相关系数都没有高于0.3，如图所示。



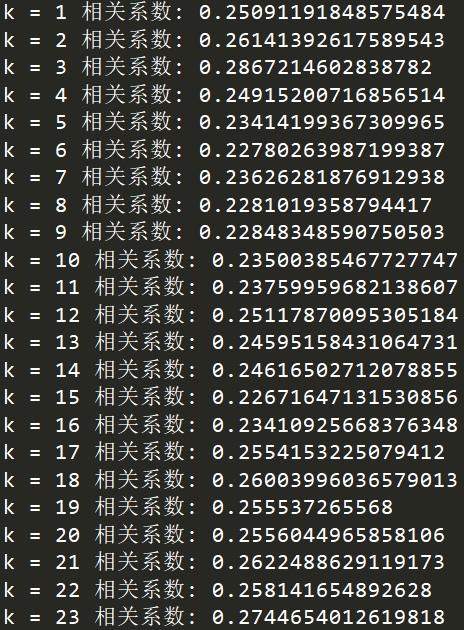
于是，我决定为训练集和测试集的数据分别生成一个TF-IDF矩阵来替代ONE-HOT矩阵，最后测试的效果有显著提高。

使用TF-IDF矩阵后，再将曼哈顿距离和欧式距离对比，试用不同的K值来选择最优。

使用曼哈顿距离时：



使用欧式距离时：

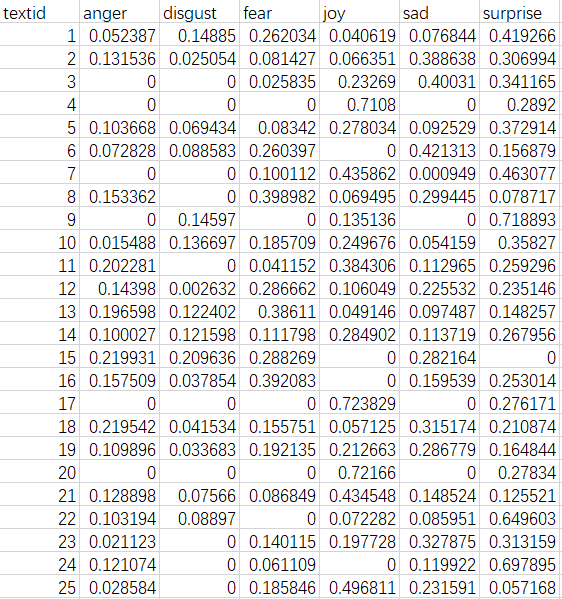


经过多次实验测试，最终发现使用曼哈顿距离，K=2时效果最佳。

* **实验结果及分析**

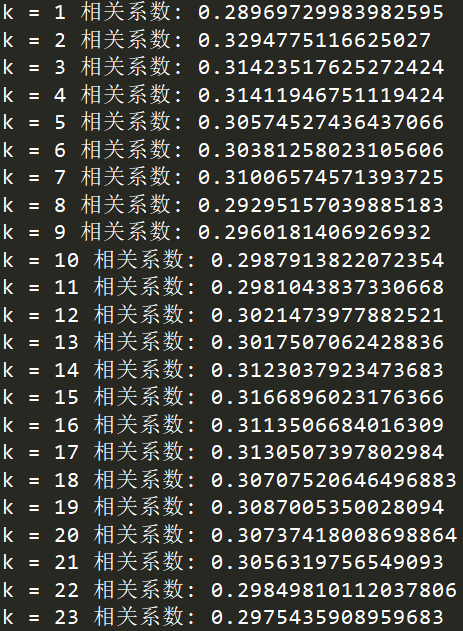
**1. 实验结果展示示例**

对测试集数据进行预测结果如图：



**2. 评测指标展示即分析**

使用曼哈顿距离，K=2时验证集结果的相关系数如图所示：



* **思考题**

**1. 为什么时倒数呢？**

因为距离越近，所对应的概率所占的比重越大。

**2. 同一测试样本的各个情感概率总和应该为1，如何处理？**

对各个情感概率进行归一化操作，使得他们的总和为1。