中山大学计算机学院人工智能本科生实验报告（2022学年春季学期）

课程名称：Artificial Intelligence

教学班级	专业（方向）	学号	姓名
2班	计算机科学与技术	21307174	刘俊杰

一、实验题目

Week 9 k-NN文本分类

二、实验内容

实验内容

在给定文本数据集完成文本情感分类训练，在测试集完成测试，计算准确率。

实验要求

1. 文本的特征可以使用TF或TF-IDF（也可以使用sklearn库提取特征）
2. 利用k-NN完成对测试集的分类，并计算准确率 (1)不可直接调用机器学习库中的KNN算法（仅可用于和自己的方法对比准确率） (2)可使用各种提取文本特征的辅助工具，如OneHotEncoder TfidfVectorizer 等

其他说明：

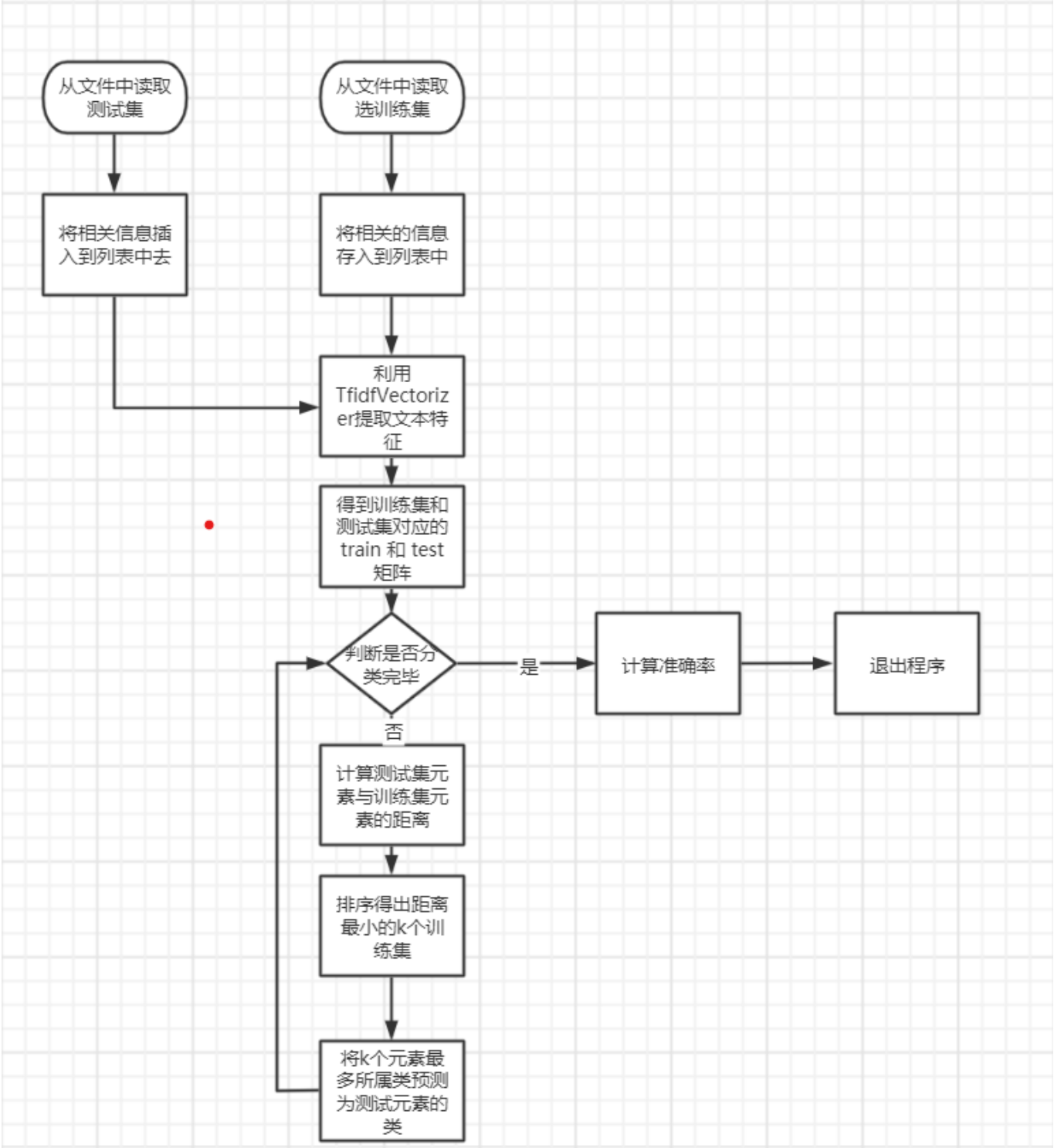
本次实验我们鼓励大家多做对比和优化，可以对比不同方法（onehot和tf-idf，k的不同取值，调库和自己实现等）或进行优化（归一化、kd树等）得到更高的准确率。 由于是多分类问题，准确率在20%以上就差不多了，最高也不会超过40%。

1. 算法原理

k-近邻分类

。给定一个训练数据集 。对新的输入实例，在训练数据集中找到与该实例最邻近的k个实例， 。这k个实例的多数属于某个类，就把该输入实例分类到这个类中。

2.流程图



3.关键代码展示

1. 读取文件，提取文本特征

```
def read_file(f):
    data=[]#存放对应的信息特征
    sentence=[]#存放文本信息
    for line in f:
        tmp=line.strip().split(" ")
        if tmp[0]=="documentId":#跳过非数据文本
            continue
        tag=int(tmp[0])#文本序号
```

```
            category=int(tmp[1])#情绪
            emotion=tmp[2]##分类标签
            sentence_tmp=''
            for i in range(3,len(tmp)):#提取每一句的单词
                sentence_tmp+=tmp[i]
                if i!=len(tmp)-1:sentence_tmp+=" "
            sentence.append(sentence_tmp)
            data.append([tag,category,emotion,sentence_tmp])
        return data,sentence#data是每一句的信息，sentence是每一句的文本
```

2. 不同的距离度量方法(使用np计算优化的能加速计算)

```
def O_distance(A,B):#计算欧氏距离
    return math.sqrt(sum([(a - b)**2 for (a,b) in zip(A,B)]))
def O_distance_np(A,B):#np优化后的计算欧氏距离
    x1 = np.array(A)
    x2 = np.array(B)
    return np.sqrt(np.sum((x1 - x2)**2))
def Man_distance(A,B):#计算曼哈顿距离
    return (sum([abs(a - b) for (a,b) in zip(A,B)]))
def Man_distance_np(A,B):#np优化后的计算曼哈顿距离
    x1 = np.array(A)
    x2 = np.array(B)
    return (np.sum(abs(x1 - x2)))
def Lp_distance(A,B,n):#计算Lp距离(闵氏距离)
    return pow((sum([pow(abs(a-b),n) for (a,b) in zip(A,B)])),1/n)
def Lp_distance_np(A,B):#np优化后的计算Lp距离(闵氏距离)
    x1 = np.array(A)
    x2 = np.array(B)
    return pow((np.sum(abs(x1 - x2)**(len(A)))),1/len(A))
```

3. TfidfVectorizer提取文本特征

```
t=TfidfVectorizer()#TfidfVectorizer提取文本特征
train=t.fit_transform(train_sentence)#读取训练集特征，此时返回一个sparse稀疏矩阵
test=t.transform(test_sentence)#读取测试集特征，此时返回一个sparse稀疏矩阵
train_matrix=train.toarray()#转换成列表
test_matrix=test.toarray()#转换成列表
```

4. k近邻分类(这里展示的时余弦相似度的k近邻分类，余弦相似度越大越好)

```
def knn1(train_set,test_set,k,train_data,test_data):#使用余弦相似度进行knn分类
    correct_num=0#统计分类正确的个数
    time_start=time.time()#记录开始时间
    m=train_set.shape[0]#训练集元素个数
    n=test_set.shape[0]#测试集元素个数
    simi=cosine_similarity(test_set,train_set)#计算余弦相似度
```

```
    for i in range(n):
        indice=np.argsort(simi[i])[-k:]#从小到大排序，获取最后k个的下标(余弦相似度越大越好)
        t=[0 for l in range(6)]#记录选取出的k个元素的中各类别占了多少元素
        distance=[0 for l in range(6)]#记录选取出的k个元素的中各类别的最小距离(余弦相似度越大越好)
        for j in range(k):
            t[train_data[indice[j]][1]-1]+=1
            distance[train_data[indice[j]][1]-1]+=simi[i][indice[j]]
        max_id=0
        max=0
        max_weight=-10000000000
        for p in range(len(t)):
            if max<t[p]:
                max=t[p]
                max_id=p
                max_weight=distance[p]
            elif max==t[p] and max_weight<distance[p]:
                max=t[p]
                max_id=p
                max_weight=distance[p]
        if max_id==test_data[i][1]-1:#若预测正确
            correct_num+=1
    time_end=time.time()#结束时间
    spend_time=time_end-time_start#花费时间
    print("距离度量方式:余弦相似度")
    print("选取k值:",k)
    print("spend time:",spend_time,"s")
    print("accuracy:",correct_num/n*100,"%")
    print("--------------------------")
```

4.创新点&优化

如果直接计算距离，那么整个程序的时间复杂度很高，于是为了加速，采用了如下创新点：(1)计算余弦相似度，一开始每次计算一个测试集元素和训练集的距离，花费时间过长，后面发现直接使用库函数得到整个测试集和训练集的余弦相似度矩阵会减少计算时间。(2)通过查询资料得到使用NumPy可以加快运算。首要原因在于NumPy的核心是经过良好优化的C代码，具有编译代码的速度。其次在于NumPy的数据结构设计和算法。 np优化前：

```
def O_distance(A,B):#计算欧氏距离
    return math.sqrt(sum([(a - b)**2 for (a,b) in zip(A,B)]))
    return np.sqrt(np.sum((x1 - x2)**2))
def Man_distance(A,B):#计算曼哈顿距离
    return (sum([abs(a - b) for (a,b) in zip(A,B)]))
def Lp_distance(A,B,n):#计算Lp距离(闵氏距离)
    return pow((sum([pow(abs(a-b),n) for (a,b) in zip(A,B)])),1/n)
```

np优化前：

```
def O_distance_np(A,B):#np优化后的计算欧氏距离
    x1 = np.array(A)
    x2 = np.array(B)
    return np.sqrt(np.sum((x1 - x2)**2))
def Man_distance_np(A,B):#np优化后的计算曼哈顿距离
    x1 = np.array(A)
    x2 = np.array(B)
    return (np.sum(abs(x1 - x2)))
def Lp_distance_np(A,B):#np优化后的计算Lp距离(闵氏距离)
    x1 = np.array(A)
    x2 = np.array(B)
    return pow((np.sum(abs(x1 - x2)**(len(A)))),1/len(A))
```

三、实验结果及分析

1. 实验结果展示示例

对于选取不同的k值和选用不用的距离度量方式得到的准确率和运行时间的结果放在了result文件中。同时也将实验结果放入评测指标中进行对比展示。

2. 评测指标展示及分析

(1)对优化前后的结果进行对比展示(这里统一选取k值为15)

距离度量方式	k	正确率	运行时间
余弦相似度(优化后)	15	37.9 %	0.04548931121826172 s
欧氏距离(未优化)	15	33.300000000000004 %	148.56199622154236 s
欧氏距离(优化后)	15	33.2 %	5.566941738128662 s
曼哈顿距离(未优化)	15	39.4 %	129.38112092018127 s
曼哈顿距离(优化后)	15	39.4 %	4.8985536098480225 s
Lp距离(未优化)	15	26.6 %	183.3200216293335 s
Lp距离(优化后)	15	26.6 %	11.006125926971436 s

(2)对使用OneHot和使用TfidfVectorizer进行对比展示(这里统一选取优化后的)

提取方式	k	欧氏距离	曼哈顿距离	Lp距离
OneHot	15	35%	35%	35%
TfidfVectorizer	15	33.2%	39.4%	26.6%
OneHot	20	34.8%	34.8%	34.8%
TfidfVectorizer	20	33.0%	38.6%	28.9%

(3)对选取不同k值进行对比展示(这里统一展示优化后的)

k	余弦相似度	欧氏距离	曼哈顿距离	Lp距离
5	34.8%	32.6%	36.1%	28.199999999999996 %
10	38.4%	35.5%	39.800000000000004 %	21.0 %
15	37.9 %	33.2%	39.4%	26.6%
17	38.0 %	33.900000000000006 %	38.7 %	28.9 %
20	37.5 %	33.0%	38.6%	28.9%

(4)将测试集和训练集反转运行对比展示

k	余弦相似度	欧氏距离	曼哈顿距离	Lp距离
15	42.68292682926829 %	43.49593495934959 %	42.27642276422765 %	30.89430894308943 %
31(k取根号N)	43.08943089430895 %	41.86991869918699 %	41.86991869918699 %	39.02439024390244 %

分析：

(1)通过展示(1)可以看出，优化后的准确率基本相同(细微的不同可能是因为运算浮点数的误差)，在时间上有很大的优化，原本一到两分钟的运行时间加速到几秒到十几秒。 (2)对使用OneHot和使用TfidfVectorizer进行对比，使用OneHot三种距离度量方式的准确率基本相同，TfidfVectorizer三种距离度量方式的准确率有差异，这是特征提取的差异，整体上OneHot三种距离度量方式较快。 (3)对选取不同k值进行对比，k值不同对准确率有一定的影响，在根号N周围取值的准确率较高，k过大或k过小效果不好。这是因为k过大：学习样本更多，会引入更多的噪音 → 可能存在欠拟合的情况；k过小：参考样本少 → 容易出现过拟合的情况。 (4)将测试集和训练集反转运行对比，发现准确率提高了许多，这是因为训练集元素个数提高了，训练集反馈给我们的提取的信息更多了，有利于我们做出更到的预测。

四、参考资料

实验课PPT 为什么NumPy这么快(https://zhuanlan.zhihu.com/p/599305311)