人工智能实验

文本数据处理与 KNN

第 1 次实验

姓名：陈家豪

班级：2018 级计科 1 班

学号：18308013

目录

1 实验任务 1：TF-IDF 矩阵的表示

1.1 算法原理

由于文本由字符、单词等不适合进行分析的元素组成，故在数据分析前，我们需要对文本进行数据处理。文本数据处理分为四个步骤：

(1) 分词；

(2) 去停用词；

(3) 建立词表；

(4) 对词语进行编码。

本次实验任务，我们将对一个文档文件进行词表的建立和编码。

目前主流的编码方式有 One-hot 编码、词频表示、tf 表示、idf 表示、TF-IDF 表示等。对于 TF-IDF 表示，每个词语的编码由两部分组成：TF 和 IDF。TF 为词频归一化后的概率表示，对于第 i 个单词，假设其在第 d 个文档中出现 $n_{i,d}$ 次，那么第 i 个单词在第 d 个文档中的数值为：

$$tf_{i,d} = \frac{n_{i,d}}{\sum\limits_{i} n_{i,d}} \tag{1}$$

其中 $\sum\limits_{i} n_{i,d}$ 是第 d 个文档的单词总数。

IDF 是逆向文档频率。假设总共有 C 篇文档，第 i 个单词在 C_i 篇文档中出现，则：

$$idf_i = \log \frac{C}{1 + C_i} \tag{2}$$

综上，对于第 j 个单词，其在第 i 篇文档中的 TF-IDF 编码为

$$tf - idf_{i,j} = tf_{j,i} \times idf_j = \frac{n_{j,i}}{\sum\limits_{j} n_{j,i}} \times \log \frac{C}{1 + C_i} \tag{3}$$

1.2 伪代码/流程图

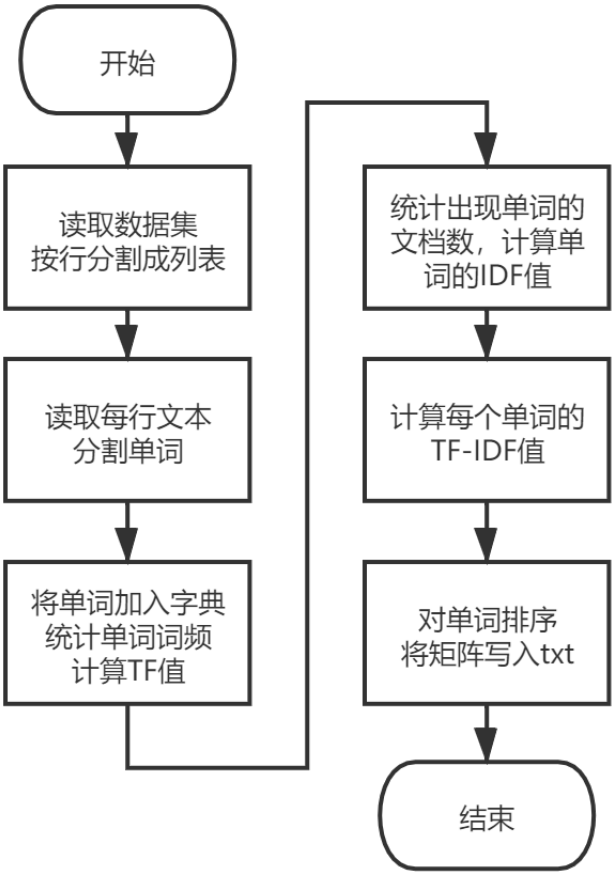

图 1: TF-IDF 矩阵计算流程图

1.3 代码截图

关键代码如图2所示。首先我们读取文本，使用 readlines() 将文本按行划分为 FileText 列表，然后使用 build_WordDict 函数计算：

```python
def build_WordDict(FileText):
    """
    构建TF-IDF字典
    """
    WordDict=dict()                 #字典
    TextSize=len(FileText)          #文档数量
    TextIndex=0
    WordNumPerSen=[0]*TextSize      #文档单词数量列表
    #构建词频字典
    for text in FileText:
        TextList=text.split('\t')   #按\t分割文本
        words=TextList[2].split()   #将句子分割成单词列表
        WordNumPerSen[TextIndex]+=len(words)
        for word in words:
            if word not in WordDict:#如果单词不在字典中
                WordDict[word]=[0]*TextSize
            WordDict[word][TextIndex]+=1
        TextIndex+=1
    #构建TF-IDF字典
    for word in WordDict:
        C_i=1
        #统计出现word的文档数
        for i in range(len(WordDict[word])):
            if WordDict[word][i]>0:
                C_i+=1
        #计算word的idf
        idf_i=math.log(TextSize/C_i)
        for i in range(len(WordDict[word])):
            #计算word的tf
            WordDict[word][i]/=WordNumPerSen[i]
            #计算word的tf-idf
            WordDict[word][i]*=idf_i
    return WordDict
```

图 2: 将文本转换为 TF-IDF 字典

build_WordDict 函数返回一个字典，字典的键为单词，对应的值为该单词的编码列表，列表元素为该单词在某个文本的编码。

最后，我们对字典的键值对按键排序，输出结果到 txt 文件即可。

1.4 实验过程与结果分析

实验结果见 “18308013_ChenJiahao_TFIDF.txt”，其中矩阵每列为按字母从小到大排序的单词对应的编码，每行为各行文本对应的编码。

以第一句文本 “mortar assault leav at least dead” 和第二句文本 “goal delight for sheva” 为例，编码结果如表1和2所示。

	mortar	assault	leav	at	least	dead
tf-idf 编码	1.07242	1.00485	0.82174	0.53931	1.00485	0.76046

表 1: “mortar assault leav at least dead” 的 tf-idf 编码

	goal	delight	for	sheva
tf-idf 编码	1.43535	1.60864	0.58298	1.60864

表 2: “goal delight for sheva” 的 tf-idf 编码

2 实验任务 2：KNN 分类任务

2.1 算法原理

KNN 是一种简单而常用的分类算法，其实现原理是在预测一个新值时，根据它距离最近（相似度最高）的 K 个点是什么类别来判断它是什么类别。本次实验中，我们需要基于训练集，对文本进行情感分类。这大致包括三个步骤：文本编码、距离计算和投票选择。

2.1.1 文本编码

由于数据集都是由单词组成的句子构成，故我们需要将文本进行编码。正如1.1所述，我们有多种方式对文本进行编码。按照助教的实验要求，我们将训练集、验证集和测试集的所有文本叠加起来，进行 TF-IDF 编码，再基于该编码计算距离。

2.1.2 距离计算

距离用于衡量句子之间的相似度。我们有两种方式表示距离：Lp 距离和余弦距离。

假设我们需要计算点 x_i 和 x_j 之间的距离，Lp 距离计算公式如下：

$$L_p(x_i, x_j) = (\sum_{l=1}^{n} |x_i^{(l)} - x_j^{(l)}|^p)^{\frac{1}{p}} \tag{4}$$

对于本次实验而言，$x_i^(l)$ 表示文本 x_i 中第 l 个单词的 TF-IDF 编码。当 $p = 1$ 时 Lp 距离即为曼哈顿距离，$p = 2$ 时 Lp 距离为欧式距离。

余弦距离的计算公式如下：

$$D_{cos} = 1 - similarity = 1 - \cos(\theta) = 1 - \frac{A \cdot B}{\|A\| \, \|B\|} = 1 - \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}} \tag{5}$$

其中 $similarity$ 为余弦相似度。

2.1.3 投票选择

在计算完毕待预测文本和训练集各文本之间的距离后，我们将该距离从小到大排序后，选取前 K 个最小距离对应的训练集文本，然后获取这些文本的标签，服从多数、选择最多的标签作为预测值。

2.1.4 根据准确率进行参数选择

在 KNN 算法中，编码方式的选择、距离度量方式和 K 值的选择都会影响算法效果的好坏。我们使用准确率衡量算法效果：

$$Accuracy = \frac{n_{correct}}{n_{total}} \tag{6}$$

其中 $n_{correct}$ 为预测正确（预测值和真实值相符）的样本数量，n_{total} 为总预测的样本数。

寻找表现最好的参数即所谓的 “调参” 过程。在本次实验中，我们需要使用验证集对不同距离度量方式和 K 值测试，选取表现最好（准确率最高）的距离度量方式和 K 值，然后再利用这两个参数对测试集预测。

以上就是 KNN 分类算法的全部过程。

2.2 伪代码/流程图

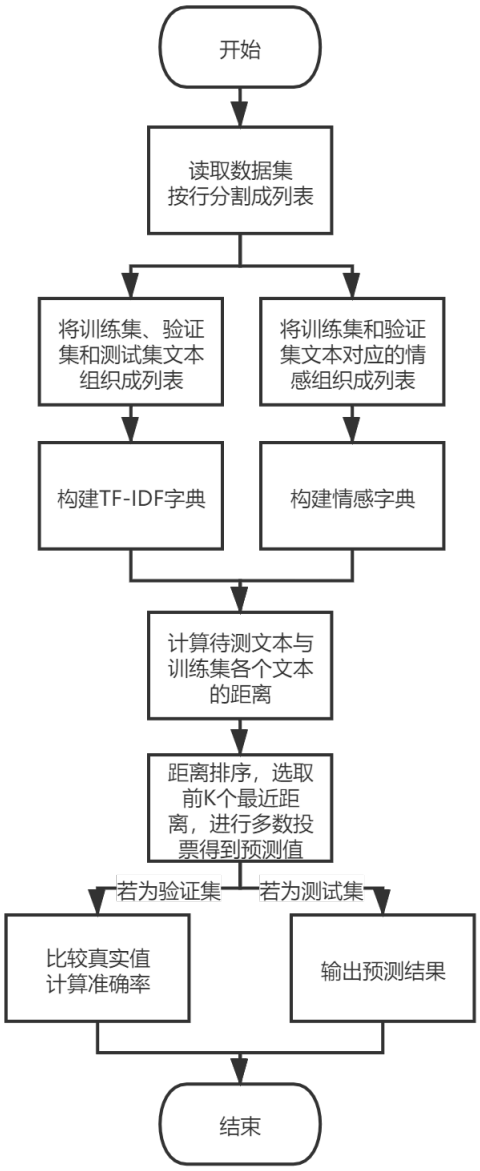

图 3: KNN 分类任务流程图

2.3 代码截图

2.3.1 文本编码

文本编码代码同4所示。与2.1.1不同的是 FileText 文本列表由训练集、验证集和测试集构成，列表元素是单纯的文本。

```python
def build_WordDict(FileText):
    """
    构建TF-IDF字典 {word:编码列表}
    """
    WordDict=dict()                    #字典
    TextSize=len(FileText)             #文档数量
    TextIndex=0
    WordNumPerSen=[0]*TextSize         #文档单词数量列表
    #构建词频字典
    for text in FileText:
        words=text.split()   #将句子分割成单词列表
        WordNumPerSen[TextIndex]+=len(words)
        for word in words:
            if word not in WordDict:#如果单词不在字典中
                WordDict[word]=[0]*TextSize
            WordDict[word][TextIndex]+=1
        TextIndex+=1
    #构建TF-IDF字典
    for word in WordDict:
        C_i=1
        #统计出现word的文档数
        for i in range(len(WordDict[word])):
            if WordDict[word][i]>0:
                C_i+=1
        #计算word的idf
        idf_i=math.log(TextSize/C_i)
        for i in range(len(WordDict[word])):
            #计算word的tf
            WordDict[word][i]/=WordNumPerSen[i]
            #计算word的tf-idf
            WordDict[word][i]*=idf_i
    return WordDict
```

图 4: 将文本转换为 TF-IDF 字典

为了方便后续计算，我们同样构建一个情感字典，字典中键为文本序号，值为该文本的情感：

```python
def build_EmotionDict(FileText):
    """
    构建emotion字典 {文档序号:emotion}
    """
    EmotionDict=dict()          #emotion字典
    TextIndex=0
    for text in FileText:
        EmotionDict[TextIndex]=text
        TextIndex+=1
    return EmotionDict
```

图 5: 构建情感字典

2.3.2 距离计算

Lp 距离计算代码如图6所示。由于训练集、验证集和测试集文本均编码存储在 WordDict 字典，故我们使用 index1 和 index2 索引待计算的两个文本的编码。

```python
def calculate_distance_p(WordDist, index1, index2, p):
    """
    计算LP距离
    """
    distance=0
    for word in WordDist.keys():
        distance+=pow(abs(WordDist[word][index1] - WordDist[word][index2]),p)
    distance=pow(distance,1/p)
    return distance
```

图 6: Lp 距离计算

同样，余弦距离计算代码如图7所示。

```python
def calculate_distance_cos(WordDist, index1, index2):
    """
    计算余弦距离
    """
    AB=0
    AA=0
    BB=0
    for word in WordDist.keys():
        AA+=pow(WordDist[word][index1], 2)
        AB+=WordDist[word][index1]*WordDist[word][index2]
        BB+=pow(WordDist[word][index2],2)
    AA=pow(AA,0.5)
    BB=pow(BB,0.5)
    distance=1-AB/(AA*BB)
    return distance
```

图 7: 余弦距离计算

2.3.3 使用验证集进行验证

验证函数如图8所示：

```python
def predict_validation
(WordDict, EmotionDict, TrainStart, TrainSize, ValidStart, ValidSize, DistanceType, K):
    """
    对验证集预测验证
    """
    correct=0       #预测正确个数
    #对每个验证集句子预测
    for i in range(ValidStart, ValidStart+ValidSize):
        distance=dict()         #该验证集句子与训练集各句的距离
        #计算该验证集句子与训练集各句的距离
        for j in range(TrainStart, TrainStart+TrainSize):
            if DistanceType==0:
                distance[j]=calculate_distance_cos(WordDict, j, i)
            else:
                distance[j]=calculate_distance_p(WordDict, j, i, DistanceType)
        #对距离进行排序
        distance_order=sorted(distance.items(),key=lambda x:x[1],reverse=False)
        prediction=dict()   #投票字典
        #选取前K个最近的距离
        for j in range(K):
            #进行投票
            if EmotionDict[distance_order[j][0]] not in prediction.keys():
                prediction[EmotionDict[distance_order[j][0]]]=1
            else:
                prediction[EmotionDict[distance_order[j][0]]]+=1
        #对投票结果进行排序
        prediction_order=sorted(prediction.items(),key=lambda x:x[1],reverse=True)
        if prediction_order[0][0]==EmotionDict[i]:#如果预测值与实际值相同
            correct+=1
        if (i-TrainSize)%20==0:
            print(str(i-TrainSize)+': done')
    return correct/ValidSize#返回准确率
```

图 8: 验证函数

由于 WordDict 包含了训练集、验证集和测试集的所有文本，故我们需要变量 TrainStart 和 ValidStart 指明训练集和验证集在 TF-IDF 字典的起始位置，TrainSize 和 ValidSize 指明训练集和验证集的样本数量。DistanceType=0 时使用余弦距离，DistanceType=i(i>0) 时使用 p=i 的 Lp 距离。我们选取不同的 DistanceType 和 K，得到相应的准确率，从而挑选表现最好的参数。

2.3.4 对测试集进行预测

预测函数如图9所示：

```python
def predict_test(WordDict, EmotionDict, TrainStart, TrainSize, TestStart, TestSize, DistanceType, K):
    """
    对测试集预测
    """
    TestEmotion=[]  #预测结果
    #对每个测试集句子预测
    for i in range(TestStart, TestStart+TestSize):
        distance=dict()
        #计算该测试集句子与训练集各句的距离
        for j in range(TrainStart, TrainStart+TrainSize):
            if DistanceType==0:
                distance[j]=calculate_distance_cos(WordDict, j, i)
            else:
                distance[j]=calculate_distance_p(WordDict, j, i, DistanceType)
        #对距离进行排序
        distance_order=sorted(distance.items(),key=lambda x:x[1],reverse=False)
        prediction=dict()   #投票字典
        #选取前K个最近的距离
        for j in range(K):
            #进行投票
            if EmotionDict[distance_order[j][0]] not in prediction.keys():
                prediction[EmotionDict[distance_order[j][0]]]=1
            else:
                prediction[EmotionDict[distance_order[j][0]]]+=1
        #对投票结果进行排序
        prediction_order=sorted(prediction.items(),key=lambda x:x[1],reverse=True)
        TestEmotion.append(prediction_order[0][0])  #将预测结果加入列表
    return TestEmotion#返回预测列表
```

图 9: 预测函数

其逻辑功能与2.3.3的验证函数一致，不同的是预测函数不再计算准确率（也无法计算），而是输出一个预测情感的列表，列表元素即为每个测试集文本的预测情感。

2.4 实验结果与分析

我们通过改变 K 值和距离度量方式, 使用验证集选取准确率表现最好的参数。实验结果如表3所示。

	余弦距离	Lp 距离		
		p=1	p=2	p=3
K=1	0.43408	0.39228	0.29904	0.26045
K=3	0.42765	0.37942	0.23794	0.20900
K=5	0.45338	0.40514	0.18971	0.20579
K=7	0.46302	0.39871	0.17685	0.16399
K=9	0.43730	0.36977	0.17042	0.16077
K=11	0.45659	0.37299	0.18328	0.17042
K=13	0.43730	0.41158	0.21222	0.17042
K=15	0.41479	0.38907	0.24437	0.16720
K=17	0.40836	0.37621	0.24116	0.17042
K=19	0.40193	0.36977	0.25402	0.17042
K=21	0.37942	0.37621	0.27331	0.16720
K=23	0.39228	0.37621	0.27331	0.15434
K=25	0.38585	0.36656	0.26367	0.15434

表 3: 不同参数下 KNN 分类准确率

对数据可视化，如图10所示。

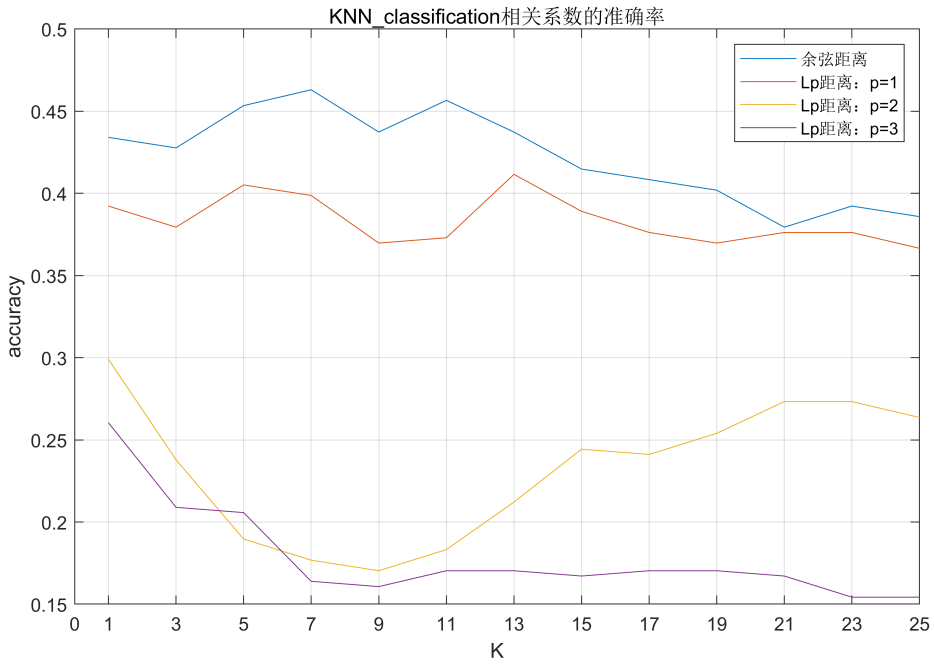

图 10: 不同参数下 KNN 分类的准确率

可见，在 K=7，余弦距离的情况下模型准确率最高，达到 46.302%。

我们选取这两个参数对测试集进行预测，预测结果见文件 “18308013_ChenJiahao_KNN_classification.csv”。以前五句文本为例，算法的预测结果如表4所示。

序号	测试集文本	预测值
1	senator carl krueger thinks ipods can kill you	surprise
2	who is prince frederic von anhalt	surprise
3	prestige has magic touch	joy
4	study female seals picky about mates	joy
5	no e book for harry potter vii	joy

表 4: 对测试集的文本预测样例

3 实验任务 3：KNN 回归任务

3.1 算法原理

KNN 算法除了可以实现分类任务，也可以实现回归任务。使用 KNN 算法实现回归时，也大致分成四个步骤：文本编码、距离计算、概率计算和根据相关系数进行参数选择。

3.1.1 文本编码

同2.1.1所示，在计算距离前，我们需要对文本进行编码。同样，我们将训练集、验证集和测试集的所有文本叠加起来，进行 TF-IDF 编码，再基于该编码计算距离。

3.1.2 距离计算

同2.1.2所示，我们有 Lp 距离（$p = 1$ 时为曼哈顿距离，$p = 2$ 时为欧式距离）、余弦距离等距离度量方式。我们可以选取其中一种距离度量方式，计算待预测样本与训练集所有样本的距离，然后计算概率。

3.1.3 概率计算

在本次实验中，数据集总共有 6 种情感，训练集中的每个文本对每种情感都有一个概率值。在对待预测文本计算各个情感的概率值时，我们将待预测文本和训练集各文本的距离进行排序，选取前 K 个最近的训练集文本，然后采用以下方式计算待预测文本的各个情感概率值：

$$P_{emotion}(test_i) = \sum_{j=1}^{K} \frac{P_{emotion}(train_j)}{d(train_j, test_i)} \tag{7}$$

其中 $P_{emotion}(test_i)$ 表示第 i 个待预测文本的 emotion 概率值，$P_{emotion}(train_i)$ 表示第 i 个与待测文本距离最小的训练集文本的 emotion 概率值，$d(train_j, test_i)$ 表示第 i 个待预测文本和第 i 个与待测文本距离最小的训练集文本的距离。

但是，这样计算后，待测文本的六个情感概率值的和并不为 1，我们需要对它们进一步修改。假设 6 种情感概率值的和为 $\sum P_{emotion}(test_i)$，那么修改后的 emotion 概率值应为：

$$P'_{emotion}(test_i) = \frac{P_{emotion}(test_i)}{\sum P_{emotion}(test_i)} \tag{8}$$

3.1.4 根据相关系数进行参数选择

不同于 KNN 分类任务，在回归任务中，我们无法直接得到一个算法的准确率，但我们可以通过比较相关系数的大小来判断模型的好坏。假设实际概率向量为 X，预测概率向量为 Y，那么两个向量的相关系数为：

$$COR(X,Y) = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{\sum_{i=1}^{n}(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \overline{X})^2 \sum_{i=1}^{n}(Y_i - \overline{Y})^2}} \tag{9}$$

相关系数越大，说明一个算法预测效果越好。

在本次实验，我们对验证集文本进行预测，得到六种情感的预测概率向量，每个向量的长度是验证集文本数量。我们分别计算每个情感的预测概率向量和实际概率向量之间的相关系数，然后对六种情感的相关系数取平均值，获得最终相关系数，以此衡量算法好坏。

同样，我们使用验证集对不同距离度量方式和 K 值测试，选取表现最好（相关系数最高）的距离度量方式和 K 值，然后再利用这两个参数对测试集预测。以上就是 KNN 回归算法的全部过程。

3.2 伪代码/流程图

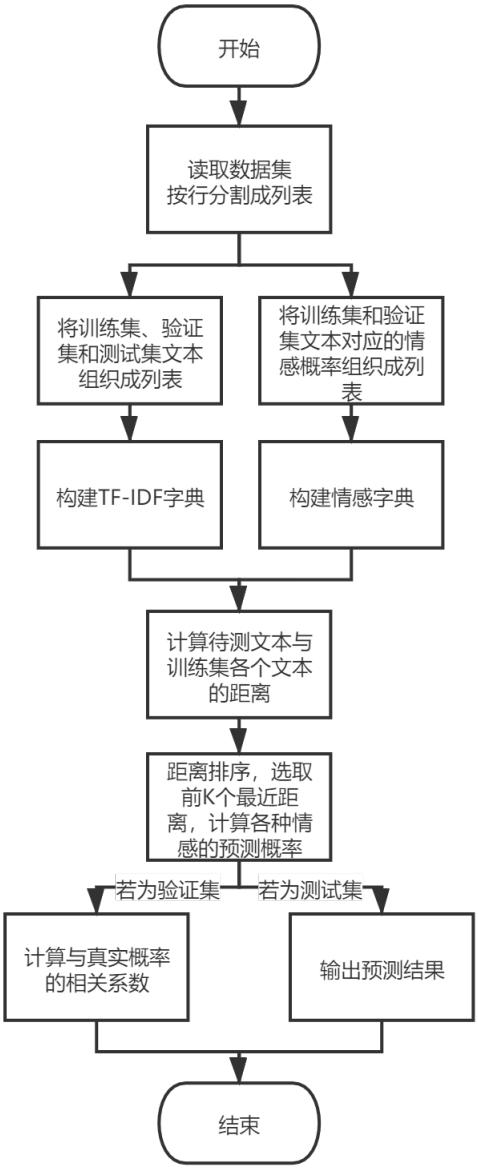

图 11: KNN 回归任务流程图

3.3 代码截图

KNN 回归任务和 KNN 分类任务十分相似，文本编码、距离计算等地方代码相同，不同的地方在于情感字典的建立和对待测文本的预测。

3.3.1 情感字典的建立

与 KNN 分类任务不同，由于数据集中每个文本对于每种情感都有相对应的概率值。故我们需要构建一种新的情感字典用于索引，键为情感序号，值为每个文本对于该情感的概率值构成的列表。

```
1  def build_EmotionDict(FileText):
2      """
3      构建emotion字典 {emotion:概率列表}
4      """
5      EmotionDict=dict()          #emotion字典
6      TextSize=len(FileText)
7      TextIndex=0
8      for text in FileText:
9          TextList=text.split(',')
10         for index in range(len(TextList)):
11             if index not in EmotionDict:
12                 EmotionDict[index]=[0]*TextSize
13             EmotionDict[index][TextIndex]=float(TextList[index])
14         TextIndex+=1
15     return EmotionDict
```

图 12: 建立情感字典

3.3.2 使用验证集进行验证并对测试集预测

KNN 回归的验证函数与 KNN 分类的验证函数基本相同，第 5 至第 18 行都是计算待预测文本与训练集各文本距离，代码如下：

```python
def predict_validation(WordDict,EmotionDict,TrainStart,TrainSize,ValidStart,ValidSize,DistanceType,K):
    """
    对验证集预测验证
    """
    PredEmotionDict=dict()
    for emotion in EmotionDict.keys():
        PredEmotionDict[emotion]=[]
    #对每个验证集句子预测
    for i in range(ValidStart, ValidStart+ValidSize):
        distance=dict()          #该验证集句子与训练集各句的距离
        #计算该验证集句子与训练集各句的距离
        for j in range(TrainStart, TrainStart+TrainSize):
            if DistanceType==0:
                distance[j]=calculate_distance_cos(WordDict, j, i)
            else:
                distance[j]=calculate_distance_p(WordDict, j, i, DistanceType)
        #对距离进行排序
        distance_order=sorted(distance.items(),key=lambda x:x[1],reverse=False)
        #对各种情感进行预测
        for emotion in PredEmotionDict.keys():
            PredEmotionDict[emotion].append(0)
            for i in range(K):
                if distance_order[i][1]==0:#如果距离为0，直接学习
                    PredEmotionDict[emotion][-1]=EmotionDict[emotion][distance_order[i][0]]
                    break
                else:#如果距离不为0，计算情感概率
                    PredEmotionDict[emotion][-1]+=EmotionDict[emotion][distance_order[i][0]]/distance_order[i][1]
        #修改情感概率，使它们和为0
        EmotionSum=0
        for emotion in EmotionDict.keys():
            EmotionSum+=PredEmotionDict[emotion][-1]
        for emotion in EmotionDict.keys():
            PredEmotionDict[emotion][-1]/=EmotionSum
        if (i-TrainSize)%20==0:
            print(str(i-TrainSize)+': done')
    #计算真实情感概率和预测情感概率的相关系数
    COR=0
    for emotion in EmotionDict.keys():
        avg_TrueVal=sum(EmotionDict[emotion][ValidStart:ValidStart+ValidSize])/ValidSize
        avg_PredVal=sum(PredEmotionDict[emotion])/ValidSize
        cov=0
        sigmaX=0
        sigmaY=0
        for i in range(ValidSize):
            X=EmotionDict[emotion][i+ValidStart]-avg_TrueVal
            Y=PredEmotionDict[emotion][i]-avg_PredVal
            cov+=X*Y
            sigmaX+=pow(X,2)
            sigmaY+=pow(Y,2)
        COR+=cov/pow(sigmaX*sigmaY,0.5)
    #计算最终相关系数
    COR/=len(EmotionDict.keys())
    return COR#返回相关系数
```

图 13: 验证函数

不同之处在于代码第 20 至 35 行，函数通过3.1.3的公式计算待测文本的各个情感概率，然后将预测结果构建成一个新字典 PredEmotionDict，键是情感序号，值是每个文本对应该情感的预测概率列表。最后我们在第 36 至 53 行计算最终相关系数并返回。

对测试集预测的函数代码与对验证集验证的函数基本相同，只是并不计算相关系数、而是直接返回预测矩阵。

3.4 实验结果与分析

我们通过改变 K 值和使用不同的距离度量方式来选取验证集相关系数表现最好的参数。实验结果如表5所示。

	余弦距离	Lp 距离		
		p=1	p=2	p=3
K=1	0.33540	0.26660	0.25888	0.17923
K=3	0.37736	0.31517	0.25843	0.24192
K=5	0.39172	0.32197	0.25737	0.22176
K=7	0.39493	0.31198	0.24436	0.24114
K=9	0.39361	0.30712	0.24244	0.23033
K=11	0.39054	0.32210	0.24909	0.23683
K=13	0.37780	0.32696	0.24949	0.23115
K=15	0.36910	0.32580	0.24858	0.21064
K=17	0.36454	0.32652	0.25937	0.21138
K=19	0.36128	0.32134	0.25039	0.21433
K=21	0.35694	0.31865	0.24624	0.21543
K=23	0.34163	0.31040	0.23997	0.21027
K=25	0.33490	0.30594	0.23316	0.21126

表 5: 不同参数下 KNN 回归相关系数

对数据可视化，如图14所示。

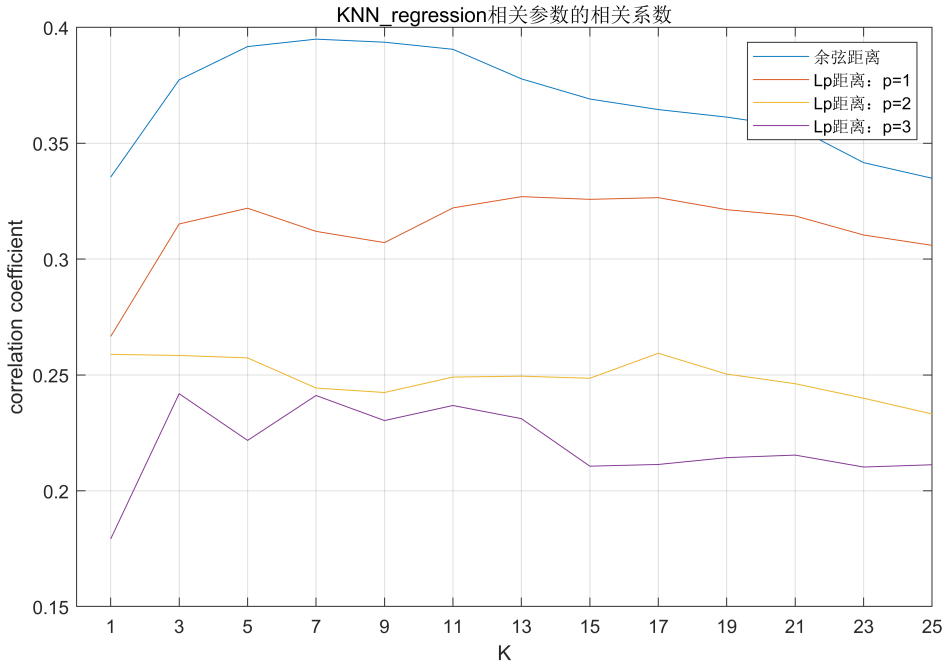

图 14: 不同参数下 KNN 回归相关系数

可见，在 K=7，余弦距离的情况下模型相关系数最高，达到 39.493%。

我们选取这两个参数对测试集进行预测，预测结果见文件 “18308013_ChenJiahao_KNN_regression.csv”。以前五句文本为例，算法的预测结果如表6所示。

序号	anger 预测值	disgust 预测值	fear 预测值	joy 预测值	sad 预测值	surprise 预测值
1	0.1357	0.182	0.0295	0.1626	0.2078	0.2823
2	0.0893	0.1271	0.0325	0.2562	0.2222	0.2726
3	0.0	0.0	0.0385	0.4151	0.1851	0.3613
4	0.0438	0.0697	0.076	0.3761	0.0913	0.3431
5	0.1067	0.0907	0.0717	0.2092	0.2702	0.2516

表 6: 对测试集的文本预测样例

4 实验总结与感想

本次实验是人工智能实验课的第一次实验，要求我们尝试对文本进行编码并使用 KNN 算法对文本进行分类、回归。以前在使用 KNN 时，我都是直接调用现有库，而这次从零搭建 KNN 算法模型给我了解更多算法细节的机会。

因为考试，本次实验完成得很仓促，没有进行一些优化（如使用 numpy 提高计算效率），也没有实现 kd 树，导致程序运行较慢。在和同学比较实验结果时，我发现在相同参数的情况下和同学的运行结果存在一些差异。经过讨论，我们认为这可能是因为选取 K 个点后进行统计时最高票票数相同，导致最终选取的数值存在差异。

总而言之，这次实验完成得比较顺利。感谢老师和助教在本次实验中提供的帮助，希望我能顺利完成下一次实验。