人工智能实验lab1 实验报告

学号：17341190 专业：计算机科学 姓名：叶盛源

TFIDF

一、算法原理

TFIDF是一种对文本进行编码的方法，在自然语言处理中我们需要将文本转换成向量矩阵的表示，才能计算文本之间的相关度或者做预测或者分类。

onehot是一种编码的方法，它先遍历所有的文本收集词语，并根据根据每个词语是否出某一篇文章中，出现的话对应向量位置为1，否则为0，但这样就忽略了词语出现的频次所带来的影响，所以就进行了改进。TF方法是通过计算这个词语在这个句子中出现的次数归一化的结果填入词向量对应的项中，公式如下：

$$tf_{i,j} = \frac{n_{k,j}}{\sum_k n_{k,j}}$$

但是还是存在一些问题，因为tf的方法容易收到一些常见词语的影响，比如一些停用词语，如：的，吗，还有一些常见但包含的能代表文本的信息少的词语，比如：转发，评论，赞这种，他们数量很多，可能会影响到一些文本关键主题的突出。所以，又引入了一个概念叫做IDF也就是反文档频率，如果这个词语在每篇文章都出现过，说明它能反应文本特性的能力不足，应该缩小它在词语向量中的权重。我们可以计算词语的反文档频率，公式如下：

$$idf_i = log\frac{|D|}{|\{j : t_i \in d_j\}|}$$

然后我们就可以将反文档频率乘到每个词语的tf上，就可以得到TF-IDF矩阵的结果

$$tfidf_{i,j} = tf_{i,j} \times idf_i$$

二、 伪代码

```
Procedure TFIDF(data)
// input: data是文本的字符串组成列表
// return: 返回TFIDF矩阵

  word_list:=count_words(data) # 根据所有文本的所有词语出现先后得到词语列表
  IDF_list:=IDF(data,word_list) # 通过IDF公式按照词语列表顺序计算idf值
  begin for i from 0 to len(data):
      record:=data[i]
      TF_dic=TF(record) # 计算每篇文本的TF值

      begin for j from 0 to len(word_list):
          if word in TF_dic:
              res[j]=TF_idc[word]*IDF_list[index]
          else
```

```
                res[j]=0
            end if
        end for
        TFIDF_met[i]:=res
    end for
    return TFIDF_met
```

三、 代码展示

本次编程采用模块化编程，分为以下多个部分：

读取文件函数和写文件函数：

```
def read_file():
    '''
    读取目标文件内容
    :return: 返回目标文件形成的列表
    '''
    data = []
    with open('../data/lab1_data/semeval.txt', 'r') as f:
        for record in f:
            data.append(record.split()[8:])
    return data


def write_file(data):
    '''
    将输出的TFIDF矩阵写入文件内
    :param data:
    :return:
    '''
    with open("TFIDF_matrix.txt",'a') as f:
        for i in data:
            for j in i:
                f.write(str(j)+" ")
            f.write("\n")
```

输入文本计算文本的词语频率，即TF值：

```
def TF(data):
    '''
    计算每篇文本词语的TF值
    :return:返回文本词语频率的字典
    '''
    total = len(data) # 统计总的词语数量
    dict={}
    for word in data:
        dict[word]=(dict.get(word,0)*total+1)/total
    return dict
```

输入文本统计词语：

```
def count_words(data):
    '''
    输入所有文本组成的列表
    按照出现先后收集所有的词语并组成列表返回
    :param data:
    :return: 返回词语按先后顺序形成的列表
    '''
    word_set=set()
    word_list=[]
    for record in data:
        for word in record:
            if word not in word_set: # 利用集合来查询速度更快，再按照先后顺序加入到列
表中
                word_list.append(word)
                word_set.add(word)
    return word_list
```

输入文本和词列表计算反文档频率：

```
def IDF(data,word_list):
    '''
    输入所有文本词语组成的列表，遍历目标词语列表并计算每个词语的IDF
    :param data:
    :return:返回目标词语列表的每一个词语对应的IDF
    '''
    IDF_list=[]
    total = len(data)
    for word in word_list:
        count=0
        for record in data:
            if word in record:
                count+=1
```

```python
        IDF_list.append(math.log(total/(count+1),2))
    return IDF_list
```

最后是主函数：

```python
def main():
    '''
    利用目标文件计算TFIDF矩阵
    :return: 返回TFIDF矩阵结果
    '''
    TFIDF_met=[]
    data = read_file() # 读取数据
    word_list=count_words(data) # 计算词语列表
    IDF_list=IDF(data,word_list) # 计算idf值
    # print(IDF_list)
    for record in data:
        res=[]
        TF_dic=TF(record)
        for index,word in enumerate(word_list):
            res.append(TF_dic.get(word,0)*IDF_list[index])
        TFIDF_met.append(res)
    write_file(TFIDF_met)
    return TFIDF_met
```

四、 实验结果以及分析：

TFIDF实验要求是通过输入文本计算每个文本的词向量的TFIDF表示，并且词语要按照出现频次先后排序。结果大致如下图：

```
1.547181392170667 1.4496876420504743 1.1855272252636149 0.7780630848181321 1.4496876420504743 1.0971081058138181 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

每一行代表一个预料，每一列是一个词语。已知要求是按照词语出现顺序统计词语列表，则可以看到第一个语料有6个词语，正好前6个词语都有值，继续观察前6个词语的IDF向量的值：

```
[9.283088353024002, 8.698125852302846, 7.11316335158169, 4.668378508908793, 8.698125852302846, 6.58264863488291, 8
 .283088353024002, 9.283088353024002, 3.364225115749407, 9.283088353024002, 7.698125852302845, 9.283088353024002, 7
 .283088353024001, 4.7912352566943275, 7.96116025813664, 9.283088353024002, 5.153805336079036, 8.283088353024002, 8
 .283088353024002, 5.823656734386705, 9.283088353024002, 7.96116025813664, 6.96116025813664, 9.283088353024002, 7
```

可以看到第一个词语因为IDF值较高导致它的TFIDF矩阵中的值相对这几个词较高。再进一步观察其他的文本计算也结果大致如上，可知实验目标达成。

五、 思考题

1. IDF的第二个计算公式中分母多了一个1是为什么：

   因为有可能出现词向量这个词语在所有文本中都没有出现过，这样就会导致分母为0。加一可以防止这种情况，在文本总量很大的时候，加一的影响不会很大。

2. IDF有什么含义，TFIDF有什么含义：

   IDF就是指词向量每个词语的反文档频率，如果这个词语在越多的篇章中出现，则这个词语就是一个常见的词语，他很大概率不能反映文本的特征，因此作为一个权重降低文本在这个词语维度上的属性值。

   TFIDF就是指将IDF的值乘在TF上后得到的结果，这个结果相当于过滤了常见的词语，保留了更能反映文本特征的属性，一般比TF或者onehot有更好的效果。

KNN分类问题

一、 算法原理

KNN算法是先通过将文本转化成向量形式，然后在所有向量中找出和他距离最近的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，用它作为目标文本的标签值，这里涉及了向量距离的计算方法，当k=1的时候是街区距离，当k=2的时候是最常用的欧式距离。

$$dist(X,Y)=\sqrt[k]{\sum_{1}^{n}(x_i-y_i)^k}$$

还有一种方法是余弦距离：

$$cos(\overrightarrow{A},\overrightarrow{B})=\frac{\overrightarrow{A}\cdot\overrightarrow{B}}{|\overrightarrow{A}||\overrightarrow{B}|}$$

$$dist(\overrightarrow{A},\overrightarrow{B})=1-cos(\overrightarrow{A},\overrightarrow{B})\geq 0$$

我们可以通过这种方法预测未知文本的标签，我们只需要将文本先转化成向量，再计算距离即可，转化成向量的方法可以采用onehot，词语频次或者上面提到的TFIDF等方法，但经过观察，因为这个数据集大多为短文本，所以最简单实用的方法就是使用onehot的方法，经过测试它在验证集上的准确度要高于TFIDF

二、 伪代码

```
Procedure KNN_classification(train_data, validation_data)
// input: train_data和validation_data是文本和它对应的标签组成的数组
// output: 返回模型在验证集上的准确度

word_list:=count_words(train_data) # 在训练集上统计所有的词语
```

```
met:=onehot(df, word_list) # 利用onehot或者TFIDF的方法训练文本的向量

predict_tag:=char[]
begin for k from 1 to 30: # 改变不同的k值看实验的效果
    met:=onehot(validation_data,word_list) # 利用onehot或者TFIDF的方法验证文本的向量

    begin for i from 0 to len(met):
        float dis_vec[len(met)]# 用来当前预料和所有训练后的语料的距离，并选取前k个最近的语料

        begin for j from 0 to len(train_met):
            dis:=cal_distance(train_met[j],met[i])# 计算两个向量的距离
            dis_vec[j]:=dis # 收集距离
        end for

        k_close:=heapq.nsmallest(k,dis_vec)# 利用最小堆的方法选取前k个最小元素标签
        label:=max(k_close,key=tag_list.count) # 求出k个标签中数量最多的标签

        predict_tag[i]:=label#收集标签

    end for

    res=cal_accuracy(predict_tag,validation_data) # 预测结果和验证集比较计算准确度
    print("KNN模型在k=%d分类问题的验证集上的准确度为：%s"%(k,res)) # 输出准确度
end for
```

三、 代码展示

读取文件模块，usecols可以选择读取csv文件的哪一列，在调用TFIDF的时候会使用到：

```
def read_file(file_name,usecols=None):
    '''
    读取csv文件数据特定的列
    :return:返回读取的csv文件
    '''
    df = pd.read_csv('../data/lab1_data/classification_dataset/'+file_name,usecols=usecols)
    return df
```

收集所有文本的所有词语：

```
def count_words(df):
```

```
    '''
    输入所有文本组成的列表
    按照出现先后收集所有的词语并组成列表返回
    :param data:
    :return: 返回词语按先后顺序形成的列表
    '''
    word_set=set()
    word_list=[]
    for i in range(len(df)):
        record = df.iloc[i][0].split()
        for word in record:
            if word not in word_set: # 利用集合来查询速度更快，再按照先后顺序加入到列表中
                word_list.append(word)
                word_set.add(word)
    return word_list
```

计算词语向量，可以用onehot也可以用TFIDF的方法：

```
def onehot(df,word_list):
    '''
    构建每个语段的onehot向量
    :return:
    '''
    onehot_met=np.zeros(shape=(len(df),len(word_list)))
    for index in range(len(df)):
        record = df.iloc[index][0].split()
        for i,word in enumerate(word_list):
            if word in record:
                onehot_met[index][i]=1
    return onehot_met

def TFIDF(df,word_list):
    '''
    调用前面写的tfidf算法来训练矩阵
    :return: 返回TFIDF矩阵
    '''
    df = read_file("train_set.csv",[0])
    data = [a[0].split() for a in np.array(df).tolist()]
    IDF_list = T.IDF(data, word_list)  # 计算idf值
    TFIDF_met=np.empty(shape=(len(data),len(word_list)))
    for i,record in enumerate(data):
        TF_dic = T.TF(record)
        for index, word in enumerate(word_list):
            TFIDF_met[i][index]= TF_dic.get(word, 0) * IDF_list[index]
    return TFIDF_met
```

计算向量的距离，cos参数可以决定是是一般的距离算法还是余弦距离，n可以选择阶数，是街区距离还是欧式距离等：

```
def cal_distance(vec1,vec2,cos=False,n=2):
    '''
    计算两个向量之间的距离，默认为欧式距离，可以调整改变阶数，cos为True是计算余弦相似度
    :return:返回一个距离值
    '''
    if not cos:
        return np.sum((vec1-vec2)**2)**0.5
    else:
        num = np.inner(vec1,vec2)   # 若为行向量则 A * B.T
        denom = np.linalg.norm(vec1) * np.linalg.norm(vec2)
        if denom==0:
            denom=0.001
        cos = num / denom  # 余弦值
        return 1-co
```

KNN模型预测，包括计算距离和利用堆的方法提取前k个元素

```
def KNN_predict(k,word_list,filename,train_met,train_df):
    '''
    读取验证集上的数据并用KNN模型计算预测结果
    :return: 返回模型预测出来的测试集上的结果
    '''
    df=read_file(filename)
    met = onehot(df,word_list) # 使用onehot的方法统计
    predict_tag=[] # 预测出来的情感标签值
    for i,record in enumerate(met):
        dis_vec=[]# 记录当前预料和所有训练后的语料的距离，并选取前k个最近的语料
        for i,train_record in enumerate(train_met):
            dis = cal_distance(record,train_record,cos=True)
            dis_vec.append((dis,i))
        heapq.heapify(dis_vec)
        k_close = heapq.nsmallest(k,dis_vec)# 利用最小堆的方法选取前k个最小元素
        tag_list=[]
        for r in k_close:
            tag_list.append(train_df.iloc[r[1]][1])
        if len(tag_list)>0:
            # 防止不存在标签
            predict_tag.append(max(tag_list,key=tag_list.count)) # 取众数计算
        else:
            predict_tag.append("null") # 若不存在标签则加入null
    return predict_tag,df
```

和验证集比较计算精确度

```
def cal_accuracy(predict_tag,validation_df):
    '''
    根据模型预测的标签结果和在验证集上真实结果比较并计算精确度
    :return: 返回精确度
    '''
    total=len(validation_df)
    count=0
    for i in range(total):
        if predict_tag[i]==validation_df.iloc[i][1]:
            count+=1
    return count/total
```

主函数：

```
def main():
    met,word_list,train_df = train()
    for k in range(1,24):
        predict_tag,validation_df =
KNN_predict(k,word_list,'validation_set.csv',met,train_df)
        res=cal_accuracy(predict_tag,validation_df)
        print("KNN模型在k=%d分类问题的验证集上的准确度为：%s"%(k,res))
```

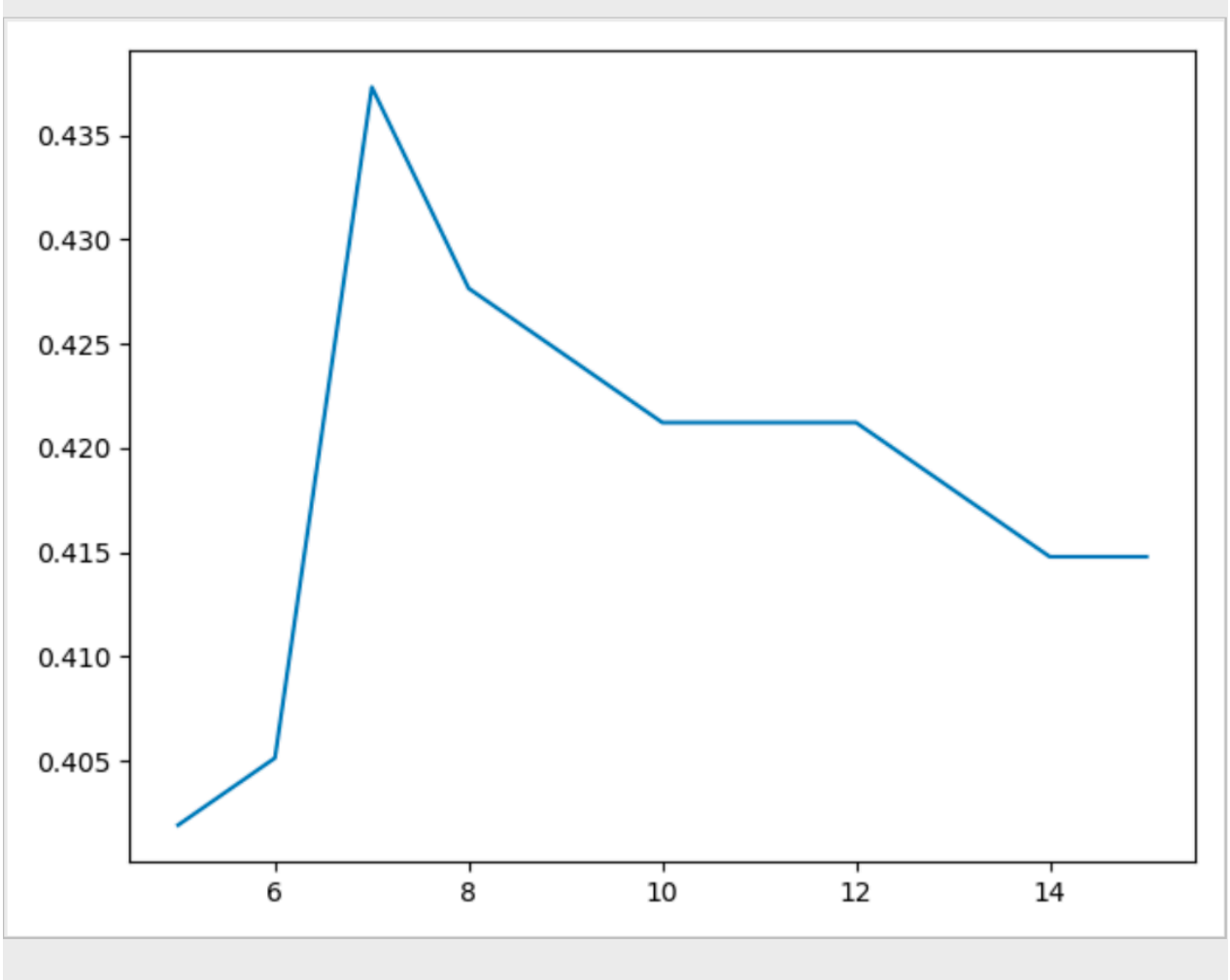
四、 创新点

在选取前k个最近元素的时候，没有采用一般的排序方法，想到了之前在数据结构课上学过的堆，和之前在leetcode上打过的类似选取k个最大元素的题目，想到了可以用堆的方法来取，于是调用了python的heapq库的方法来取前k个元素，经过和排序算法的比对测试，运行速度速度有很大的提升。

五、 实验结果和分析

结果展示和分析

我们通过遍历不同的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值和它对应的在验证集上的精确度表现情况。经过测试，这里我们取k从1到29，依次输出它的结果并将k值和对应的精确度作出折线图，如下：



可以看到准确度会随着k的值变化，可能因为训练和验证样本数量比较少，没有呈现一个很明显的变化趋势，总体上看是先上升后下降，可以预测到当数据量比较大的时候可能k值在某一个值会有很好的表现，我们可以取那个最大精确度时的k值来作为模型参数，进行以后对未知数据集的预测工作。对p值的改变也一样，使用不同的p值，选取精确对最优的情况下的p值作为模型的参数来进行预测。

其中k在大约7的时候表现最好，达到了43.7%的准确度

```
KNN模型在k=5分类问题的验证集上的准确度为: 0.40192926045016075
KNN模型在k=6分类问题的验证集上的准确度为: 0.40514469453376206
KNN模型在k=7分类问题的验证集上的准确度为: 0.43729903536977494
KNN模型在k=8分类问题的验证集上的准确度为: 0.42765273311897106
KNN模型在k=9分类问题的验证集上的准确度为: 0.42443729903536975
KNN模型在k=10分类问题的验证集上的准确度为: 0.4212218649517685
KNN模型在k=11分类问题的验证集上的准确度为: 0.4212218649517685
KNN模型在k=12分类问题的验证集上的准确度为: 0.4212218649517685
KNN模型在k=13分类问题的验证集上的准确度为: 0.4180064308681672
```

模型性能展示和分析

我们可以通过调节一些参数或者改变一下距离或者词向量的计算方法来进行优化

	余弦距离	欧式距离	onehot	TFIDF	精确度	最优k值
初始	0	1	1	0	42.12%	13
优化1	1	0	1	0	43.73%	7
优化2	1	0	0	1	27.65%	13
优化3	0	1	0	1	18.9%	13
最优结果	1	0	1	0	43.73%	7

可以看到TFIDF在这个短文本的数据集上表现并不好，和onehot的方法比起来精确度差了很多，而在采用了余弦相似度计算距离后精确度又了一定的提升。

这里没有展示不同的p值情况下的精确度差别，因为对于onehot不同的p值并不会改变实验的结果，因为1的无数次方都是1，0的无数次方都是0，所以没有效果。对于TFIDF的测试结果和原来的并没有很大提升，并且TFIDF的效果在这个数据集上效果并不好，这也是调节p值结果效果不明显的原因。

KNN回归问题

一、 算法原理

算法原理和分类问题一样，也是先将文本转化成词语向量后，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：

$$P(test1 is happy) = \frac{train1\ probility}{d(train1, test1)} + \frac{train2\ probility}{d(train2, test1)} + \frac{train3\ probility}{d(train3, test1)} + \cdots$$

这里最后每个文本都可以算出六个情感的概率值，我们在对验证集进行验证的时候，需要用到相关系数来进行计算，计算两个向量之间的相关性，最后将验证集所有文本的相关度加起来求平均值，就可以得到在验证集上的表现情况，相关度的公式如下：

$$r(X, Y) = \frac{Cov(X, Y)}{\sqrt{Var[X]Var[Y]}}$$

二、 伪代码

```
Procedure KNN_regression(train_data, validation_data)
// input: train_data和validation_data是文本和它对应的标签组成的数组
// output: 返回模型在验证集上的准确度

word_list:=count_words(train_data) # 在训练集上统计所有的词语
met:=onehot(df, word_list) # 利用onehot或者TFIDF的方法训练文本的向量

predict_tag:=char[]
begin for k from 1 to 30: # 改变不同的k值看实验的效果
```

```
    met:=onehot(validation_data,word_list) # 利用onehot或者TFIDF的方法验证文本的向量

    begin for i from 0 to len(met):
        float dis_vec[len(met)]# 用来当前预料和所有训练后的语料的距离，并选取前k个最近的语料

        begin for j from 0 to len(train_met):
            dis:=cal_distance(train_met[j],met[i])# 计算两个向量的距离
            dis_vec[j]:=dis # 收集距离
        end for

        k_close:=heapq.nsmallest(k,dis_vec)# 利用最小堆的方法选取前k个最小元素标签
        char emotion_list[6] # 保存每个情感计算的值

        count:=0
        begin i for from 0 to 5:

            begin j for from 0 to k:
                count:=count+k_close[j][i]/dis # 遍历最近的k个找到对应的情感然后除以距离后加和
            end for

            emotion_list[i]:=count
        end for

        predict_tag[i]:=emotion_list#收集标签

    end for

    res=cal_accuracy(predict_tag,validation_data) # 预测结果和验证集比较计算相关度
    print("KNN模型在k=%d回归问题的验证集上的准确度为：%s"%(k,res)) # 输出准确度
end for
```

三、 代码展示

除了以下几个板块其他和分类问题的KNN代码相同。

归一化，对计算结果归一化，让和为1

```
def normalize(target_list):
    '''
    将目标的列表中每一个元组或列表进行归一化处理，使他们和为1
    :return:返回归一化后的列表
    '''
    res=np.empty(shape=(len(target_list),len(target_list[0])))
    for i,record in enumerate(target_list):
        total=sum(record)
        for j,n in enumerate(record):
            res[i][j]=n/total
    return res
```

KNN预测结果

```
def KNN_predict(k,word_list,filename,train_onehot_met,train_df):
    '''
    读取验证集上的数据并用KNN模型计算预测结果
    :return: 返回模型预测出来的测试集上的结果
    '''
    df=read_file(filename)
    onehot_met = onehot(df,word_list)
    predict_tag=[] # 预测出来的情感标签值 每一个都包含 anger disgust fear joy sad
surprise
    for i,record in enumerate(onehot_met):
        dis_vec=[]# 记录当前预料和所有训练后的语料的距离，并选取前k个最近的语料
        for i,train_record in enumerate(train_onehot_met):
            dis = cal_distance(record,train_record,n=2,cos=True)
            dis_vec.append((dis,i))
        heapq.heapify(dis_vec)
        k_close = heapq.nsmallest(k,dis_vec)# 利用最小堆的方法选取前k个最小元素

        emotion_list = []
        for i in range(6):
            # 对当前语料轮流计算每一个emotion的值
            count=0
            for r in k_close:
                dis = r[0]
                if r[0]==0:
                    dis=0.001# 如果语段完全相同 距离为0，则取距离为0.001
                count+=train_df.iloc[r[1]][i+1]/dis
            emotion_list.append(count)
        predict_tag.append(emotion_list)
    predict_tag = normalize(predict_tag)
    return predict_tag,df
```

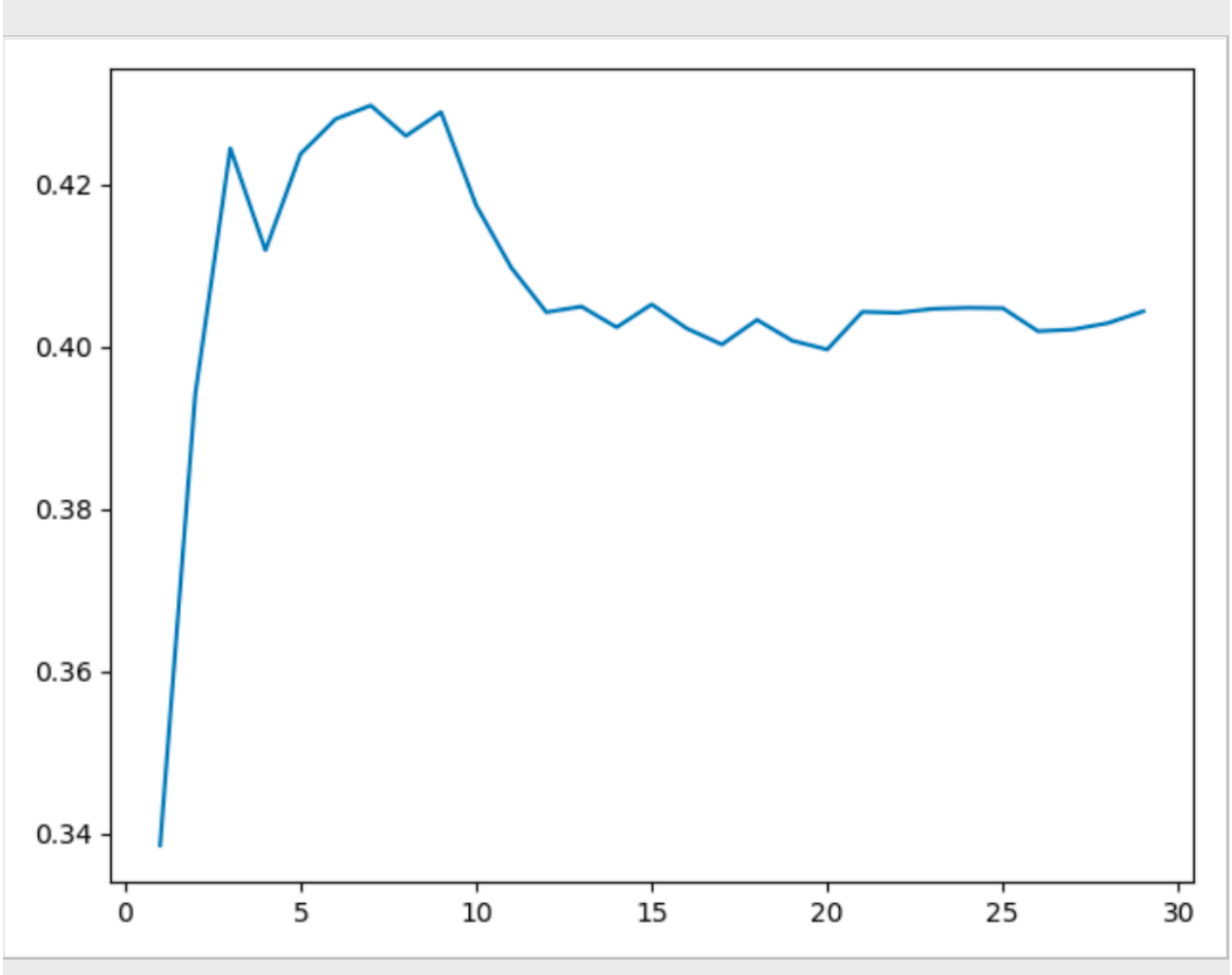
使用相关系数计算精确度

```
def cal_accuracy(predict_tag):
    '''
    根据模型预测的标签结果和在验证集上真实结果比较并计算精确度
    :return: 返回精确度
    '''
    target_df = np.array(read_file("validation_set.csv",usecols=
[1,2,3,4,5,6]))
    count=0
    for i in range(target_df.shape[0]):
        count += np.corrcoef(predict_tag[i],target_df[i])[0][1]
    return count/target_df.shape[0]
```

四、 实验结果和分析

结果展示和分析

我们通过遍历不同的k值，循环调用KNN回归算法，就可以找到在这个模型下训练集上表现最好时候的k值和它对应的在验证集上的相关性表现情况。经过测试，这里我们取k从1到29，依次输出它的结果并将k值和对应的相关性作出折线图，如下：



可以看到整体也是先上升在下降的趋势。其中在k=9的时候表现的最好，接近了43%

KNN模型在k=5回归问题的验证集上的相关度为：0.4238188460970131
KNN模型在k=6回归问题的验证集上的相关度为：0.4281267949189401
KNN模型在k=7回归问题的验证集上的相关度为：0.42979384127933495
KNN模型在k=8回归问题的验证集上的相关度为：0.42603543982405684
KNN模型在k=9回归问题的验证集上的相关度为：0.42897731802530464
KNN模型在k=10回归问题的验证集上的相关度为：0.41748065513269084

模型性能展示和分析

我们可以通过调节一些参数或者改变一下距离或者词向量的计算方法来进行优化

	余弦距离	欧式距离	onehot	TFIDF	相关性	最优k值
初始	0	1	1	0	36.92%	14
优化1	1	0	1	0	43.0%	9
优化2	1	0	0	1	10.48%	19
优化3	0	1	0	1	10.39%	19
最优结果	1	0	1	0	43.0%	9

可以看到TFIDF在这个短文本的数据集上表现并不好，和onehot的方法比起来精确度差了很多，而在采用了余弦相似度计算距离后相关性又了一定的提升。

总体的实验效果和用KNN在分类问题上的表现相似，都是TFIDF表现没有onehot好，并且用余弦相似度后有一定相关性的提升，效果比较明显

五、 思考题

1. 为什么是倒数呢?

   因为距离越远理论上和当前文本的相似程度就越小，则它作为权重削减它的情感对当前文本情感的影响力，所以需要取倒数，分母越大，数越小。

2. 同一测试样本的各个情感概率总和应该为一 如何处理?

   可以做一个归一化处理，将所有情感概率加起来的到一个total，然后每个情感概率除以这个total作为归一化后的情感概率。