中山大学数据科学与计算机学院

移动信息工程专业-人工智能

本科生实验报告

（2017-2018 学年秋季学期）

课程名称：**Artificial Intelligence**

教学班级		专业（方向）	**移动互联网**
学号		姓名	**Jw**

一、 实验题目

K 近邻与朴素贝叶斯 -------- 分类和回归

二、 实验内容

1. 算法原理

K 近邻：

K 近邻算法是数据挖掘分类与回归算法中最简单的一种。首先我们假定给出一篇包含 n 篇文本的训练集，我们将每个文本用一个特征向量表示，这个特征向量的每一维表示一个单词，常用的求特征向量的方法有 onehot，tf，和 tfidf。然后对每一个测试文本，求它与每一个训练文本的特征向量之间的距离(可以是欧式距离、曼哈顿距离或余弦距离)。在

分类问题上，如果一个样本在特征空间中的 K 个最相邻的样本中的大多数属于某个类别，则该样本也属于这个类别。即:

test1 $label$ = the most common（{label | label belong to the Top K nearest trains}）

在回归问题上，训练集样本不是单纯地打上标签，而是知道它属于每种标签的概率(概率和为 1)，那么对于每个测试集样本，取它在特征空间中的 K 个最相邻的样本的距离（余弦距离为正且值越大为越相近，为负且值越小为越远，因为是根据两个向量之间的夹角看的），将距离的倒数作为权重，计算属于每个标签的概率。例如:

$$\mathrm{P(test1\ is\ label\_i)} = \frac{\mathrm{p(train1\ is\ label\_i)}}{\mathrm{d(train1, test1)}} + \frac{\mathrm{p(train2\ is\ label\_i)}}{\mathrm{d(train2, test1)}} + \cdots$$

补充知识：

- 距离公式：

  L_p距离(所有距离的总公式)：

  - $\mathrm{L_p}(x_i, x_j) = \left\{\sum_{l=1}^{n} \left|x_i^{(l)} - x_j^{(l)}\right|^p\right\}^{\frac{i}{p}}$

- $p = 1$：曼哈顿距离；
- $p = 2$：欧式距离，最常见。

余弦相似度：

$\cos\left(\vec{A},\vec{B}\right)=\frac{\vec{A}\cdot\vec{B}}{|\vec{A}||\vec{B}|}$，其中 $\vec{A}$ 和 $\vec{B}$ 表示 两个文本特征向量；

- 余弦值作为衡量两个个体间差异的大小的度量
- 为正且值越大，表示两个文本差距越小
- 为负代表差距越大，请大家自行脑补两个向量余弦值。

结果影响因素：不同的 K 值，一般取 sqrt（样本数）以内；

不同的距离度量方式

对权值或者特征向量归一化：

Name	Formula	Explain
Standard score	$X'=\frac{X-\mu}{\sigma}$	μ is the mean and σ is the standard deviation
Feature scaling	$X'=\frac{X-X_{min}}{X_{max}-X_{min}}$	X_{min} is the min value and X_{max} is the max value

朴素贝叶斯(Na ïve Bayes):

用朴素贝叶斯去解决分类和回归问题首先要知道贝叶斯定理，贝叶斯定理主要是提供了一个算法解决这样一个问题:已知某条件概率，如何求事件交换后的条件概率，即已知 P(A|B)，求 P(B|A)。定理如下:

$$P(B|A)=\frac{P(A|B)P(B)}{P(A)} \qquad \text{其中 } P(A|B)=\frac{P(AB)}{P(B)}$$

上式中，P(A|B)表示事件 B 已经发生的前提下事件 A 发生的概率，P(AB)表示 A 和 B 同时发生的概率。而当 A 和 B 两个事件是互斥的，完全独立的没有交集的两件事，此时 P(AB)=P(A)P(B)，这时就是朴素的

分类问题:朴素贝叶斯求解分类问题就是对于给出的待分类项，算出它属于每个标签的概率，哪个概率最大，就是哪个标签。类似 KNN，首先我们要求出每个样本的特征向量 x，对给定的样本特征 x，样本属于类别 ei 的概率是

$$P(ei|x)=\frac{P(x\mid ei)P(ei)}{P(x)}$$

假设 x 的维数是 K，由于是朴素的，特征条件互斥，根据全概率公式有：

$$P(ei|x)=\frac{P(ei)\prod_{i=1}^{K}p(xi\mid ei)}{P(x)}$$

由于对于同一个文本来讲，P(x)是一样的，所以预测它属于哪个分类，只要算上式的分子部分即可。对于一个给定的测试样本，分子部分常用的有两种模型：

1.伯努利模型(Bernoulli Model)：

$$P(xi|ei)=\frac{N_{ei}(xi)}{N_{ei}} \qquad P(ei)=\frac{N_{ei}}{N}$$

其中N_{ei}表示情感 ei 出现的训练集样本数，$N_{ei}(xi)$ 表示 xi 出现在了多少个情感为 ei 的样本中。N 表示训练集总样本数。不难发现这个模型中 x 向量是一个二进制数

2.多项式模型(Multinomial Model)

$$P(xi|ei)=\frac{NW_{ei}(xi)}{NW_{ei}} \qquad P(ei)=\frac{N_{ei}}{N}$$

其中，$NW_{ei}(xi)$表示在训练集中，情感为 ei 的文章中，单词 xi 出现了多少次(一篇文章中出现多次按多次算), NW_{ei}表示训练集中情感为 ei 的文章包含多少个单词(不去重)

回归问题: 与 KNN 中一样的是同样结果要输出每种情感的概率。它的特征向量为当前文本的词频，即特征空间应处理成 TF 矩阵。在回归问题中，同样只需要关注P(ei|x)如何计算.设训练集中第 j 篇文本为 dj，特征向量维数为 K,则P(ei|x)的计算方法如下:

$$P(ei|x) = \sum_{j=1}^{N} p(ei|dj) \prod_{k=1}^{K} p(xk|ei,dj)$$

$$p(xk|ei,dj) = \frac{count(xk)}{\sum_{k=1}^{K} count(xk)}$$

这里，如果测试集中出现了训练集没有的单词，我们选择忽略。最后算完需要将一个测试文本的情感概率归一化，保证概率和为 1

补充知识：**平滑**。当测试文本中存在一个词 x，x 是训练集中的单词，但是 x 没有出现在当前 dj 文本，那么$p(xk|ei,dj)$会=0 从而导致$p(ei|dj)\prod_{k=1}^{K} p(xk|ei,dj)$=0，此时，我们可以通过平滑解决。

回归：

$$p(xk|ei,dj) = \frac{count(xk)+alpha}{\sum_{k=1}^{K} count(xk)+K*alpha}$$

分类：

伯努利模型：$P(xi|ei) = \frac{N_{ei}(xi)+alpha}{N_{ei}+2*alpha}$

多项式模型：$P(xi|ei) = \frac{NW_{ei}(xi)+alpha}{NW_{ei}+alpha*N}$

以回归的平滑为例，不难看出对于测试集 x，$p(xk|ei,dj)$的分母为$\sum_{k=1}^{K} count(xk) + K * alpha$保持不变，而分子，若 count(xk)为 0，则为 alpha，否则为 count(xk)+alpha, 这样保证了$\sum_{k=1}^{K}$（$count(xk) + alpha$）$= \sum_{k=1}^{K} count(xk) + K * alpha$，即向量中各维所表示的单词的概率和为 1，同时避免了乘上了 1 个概率为 0 的数导致最后结果为 0

在上面的式子中，0<alpha<=1，当 alpha=1 时，称为拉普拉斯平滑(Laplace)；当 0<alpha<1 时称为 Lidstone 平滑

2. 伪代码

KNN 分类

Step1.	读入测试集、训练集、验证集的数据
Step2.	选择进行验证集调参还是对测试集进行预测，若选择调参则进行 Step3,否则进行 Step8;无论选择哪一步，均需先指定三个参数: MatrixType(特征空间的矩阵类型)：onehot、TFIDF Standard(特征向量类型):original(原始)、normalization(Z-Score 标准化) Distype(距离类型)：Manhattan、Euclidean、Cosine
Step3.	根据 MatrixType 的到对应的矩阵，onehot 的单词集合包括训练集和验证集所有不重复的单词，TFIDF 的单词集合只算训练集的单词
Step4.	一个循环尝试不同的 k 值，对于某个 k 值 ki，计算测试数据与各个训练数据

之间的距离，

Step5. 排序后选取距离最小的 ki 个点（若是余弦距离则选取距离最大的 ki 个点）

Step6. 确定前 ki 个点所在类别的出现频率，选择 ki 个点中出现频率最高的类别作为测试数据的分类

Step7 对于当前 ki，求当前 ki 下的准确率，然后跳回 Step4 继续使用下一个 ki 值去做预测，若循环结束则跳到 Step9

Step8 使用通过验证集得到的最优的参数 K、MatrixType 以及 Standard 去做预测，方法更预测验证集时一样，不再累述。预测完将结果输出，不用算准确率。

Step9 程序结束

KNN 回归

流程跟 KNN 分类时一样，其他不再累述，唯一不同的地方只有下面这两步

Step6:.使用最近的 K 个点去算测试数据是每种分类的概率，假设最近的 K 个文本的编号为 1-k 则使用非余弦距离时计算公式如下：

$$P(\text{test is label\_i}) = \frac{p(\text{train1 is label\_i})}{d(\text{train1, test})} + \frac{p(\text{train2 is label\_i})}{d(\text{train2, test})} + \cdots + \frac{p(\text{traink is label\_i})}{d(\text{traink, test})}$$

计算时注意，距离 d 即分母为 0，说明测试文本和训练文本一模一样，此时训练文本的各项感情的概率直接使用该训练文本的概率即可。

若使用余弦距离则公式如下：

$$P(\text{test is label\_i}) = p(\text{train1 is label}_i) * d(\text{train1, test}) + \cdots + p(\text{traink is label}_i) * d(\text{traink, test})$$

计算时如果遇到距离 d 为 1 的，明测试文本和训练文本一模一样，此时训练文本的各项感情的概率直接使用该训练文本的概率即可。最后需要对情感各概率归一化

Step7 预测完之后，将结果输出到文件中。(使用相关性验证的 excel 文档去做相关性计算)

NB 分类

Step1. 读入测试集、训练集、验证集的数据，并计算 p(ei)

Step2. 选择要预测的数据（测试集 or 验证集）、要计算的模型（默认多项式模型）以及 alpha 值(默认 1)

Step3. 根据模型先计算P(xi|ei)的分子部分 num，以及分母部分 den，存在数组里

Step4. 遍历每一个需预测文本 test[i]

初始化该文本所有情感概率 p[j]为 1

遍历每一个情感 lable[j]

遍历 test[i]的每一个单词 word

如果 word 不是训练集中的词就忽略

否则根据选择的模型加上平滑计算概率P(word|ei)

P[j] *= P(word|ej)

P[j] *= p(ej)

对 P 作归一化保证概率和为 1

Step5. 选择 p 中概率最大的情感输出

NB 回归

Step1. 读入测试集、训练集、验证集的数据，并计算 p(ei)

Step2. 选择要预测的数据（测试集 or 验证集）、以及 alpha 值(默认 1)

Step3. 遍历每一个需预测文本 test[i]

初始化该文本所有情感概率 p[j]为 0

遍历每一个情感 lable[j]

遍历每一个训练文本 train[k]

temp = 1.0

遍历 test[i]的每一个单词 word

如果 word 不是训练集中的词就忽略

否则加上平滑计算概率P(word|e，train[k])

temp *= P(word|e，train[k])

temp *= p(ej|train[k])

P[j] += temp

对 P 作归一化保证概率和为 1

Step5. 输出所有预测文本的所有情感的概率

3. 关键代码截图（带注释）

KNN：

1.标准化。根据公式对每行进行 Z-Score 标准化 x = （x-均值 mean） / 标准差 std 由于标准差有可能为 0，这里我的解决办法是在分母加上一个极小的数字 1e-10 最后变成单位向量并通过将矩阵的值减去负数而变成全整数。选择 Z-Score 的原因是可以将数据正太分布，使“近的更近，远的更远”。

```
def Normalize(self, Matrix): #Z-SCORE
    n = Matrix.shape[0]
    for i in range(n):
        #求标准差tstd, 均值tmean
        tstd, tmean = Matrix[i, :].std(), Matrix[i, :].mean()
        #标准化
        Matrix[i, :] = (Matrix[i, :] - tmean)/(tstd + 1e-10) #加上1e-10防止标准差为0
        #变成单位向量
        len = np.linalg.norm(Matrix[i, :])
        if len > 1e-6:
            Matrix[i, :] = Matrix[i, :]/len
    Matrix = Matrix-np.min(Matrix)
    return Matrix
```

2. 计算 onehot 矩阵

```
def getOnehot(self, standard, train_set, test_set):# 求训练集和测试集的onehot矩阵
    word_set = list(train_set[3] | test_set[3])#单词集合为两者单词集合的并集
    train_matrix, test_matrix = [], []
    train_len, test_len = len(train_set[0]), len(test_set[0])
```

```
    #计算onehot
    for i in range(train_len):
        train_matrix.append([])
        for word in word_set:
            train_matrix[i].append(1.0 if word in train_set[0][i] else 0.0)
    for i in range(test_len):
        test_matrix.append([])
        for word in word_set:
            test_matrix[i].append(1.0 if word in test_set[0][i] else 0.0)
    train_matrix = np.array(train_matrix, dtype=float)
    test_matrix = np.array(test_matrix, dtype=float)
    #若选择了标准化，则对onehot矩阵进行标准化
    if standard == "normalization":
        train_matrix = self.Normalize(train_matrix)
        test_matrix = self.Normalize(test_matrix)
    return [train_matrix, test_matrix]
```

3. 计算 TFIDF 矩阵

```
def getTFIDF(self, standard, train_set, test_set):# 求训练集和测试集的TFIDF矩阵
    word_set = list(train_set[3])
    train_matrix, test_matrix = [], []
    train_len, test_len = len(train_set[0]), len(test_set[0])
    #计算IDF值
    IDF = dict()
    for i in range(train_len):
        s = set(train_set[0][i])
        for x in s:
            if x not in IDF:
                IDF[x] = 1.0
            else:
                IDF[x] += 1.0
    for x in IDF:
        IDF[x] = log(train_len/(1.0+IDF[x]))
    #计算训练集的TFIDF
    for i in range(train_len):
        train_matrix.append([])
        wordnum = 0
        for word in word_set:
            if word in train_set[0][i]:
                cnt = train_set[0][i].count(word)
                wordnum += cnt
                train_matrix[i].append(cnt*IDF[word])
            else:
                train_matrix[i].append(0.0)
        for j in range(len(train_matrix[i])):
            train_matrix[i][j] = train_matrix[i][j] / wordnum
```

```
    #计算需预测集的TFIDF
    for i in range(test_len):
        test_matrix.append([])
        wordnum = 0
        for word in word_set:
            if word in test_set[0][i]:
                cnt = test_set[0][i].count(word)
                wordnum += cnt
                test_matrix[i].append(cnt*IDF[word])
            else:
                test_matrix[i].append(0.0)
        for j in range(len(test_matrix[i])):
            if wordnum > 0:
                test_matrix[i][j] = test_matrix[i][j] / wordnum
            else:
                test_matrix[i][j] = 1e-10

    train_matrix = np.array(train_matrix, dtype=float)
    test_matrix = np.array(test_matrix, dtype=float)
    if standard == "normalization":
        train_matrix = self.Normalize(train_matrix)
        test_matrix = self.Normalize(test_matrix)
    return [train_matrix, test_matrix]
```

4. 计算两个向量的不同距离

```
def getdis(self, disType, v1, v2):
    if disType == "Manhattan":
        return sum(abs(v1-v2))
    elif disType == "Euclidean":
        return np.linalg.norm(v1-v2)
    elif disType == "Cosine":
        num = float(np.sum(v1*v2))
        denom = np.linalg.norm(v1)*np.linalg.norm(v2)
        cos = num/denom
        return cos
```

5. KNN 回归的概率计算

```
def calcLabel(self, testline, train_matrix, disType):
    dis = dict()
    label = [0.0 for i in range(self.label_count)]
    #计算当前测试文本testline与每一个训练文本的距离
    for i in range(train_matrix.shape[0]):
        dis[i] = self.getdis(disType, testline, train_matrix[i, :])
    #余弦距离
    if disType == "Cosine":
        dis = sorted(dis.items(), key=lambda x: (-x[1], x[0]))
        for i in range(self.K):
            if abs(dis[i][1] - 1) < 1e-6:#余弦距离为1，则测试文本与该训练文本一模一样
                label = self.train_set[1][dis[i][0]]
                break
            else:#其他情况按公式计算概率
                for j in range(self.label_count):
                    label[j] += self.train_set[1][dis[i][0]][j]*dis[i][1]
        #若概率全为0则赋成1/6
        if abs(sum(label) - 1) < 0:
            label = [1.0/6.0 for i in range(6)]
    else:#非余弦距离
        dis = sorted(dis.items(), key=lambda x: (x[1], x[0]))
        for i in range(self.K):
            if dis[i][1] < 1e-6: #距离为0，则测试文本与该训练文本一模一样
                label = self.train_set[1][dis[i][0]]
                break
            else:#其他情况按公式计算概率
                for j in range(self.label_count):
                    label[j] += self.train_set[1][dis[i][0]][j]/dis[i][1]
    Sum = sum(label)
    label = [x/Sum for x in label] #概率归一化
    return label
```

6. KNN 分类的概率计算

```
def calcLabel(self, testline, train_matrix, disType):
    dis = dict()
    #计算当前测试文本testline与每一个训练文本的距离
    for i in range(train_matrix.shape[0]):
        dis[i] = self.getdis(disType, testline, train_matrix[i, :])
    if disType == "Cosine":#余弦距离按从大到小排序
        dis = sorted(dis.items(), key=lambda x: (-x[1], x[0]))
    else:#其余距离按从小到大排序
        dis = sorted(dis.items(), key=lambda x: (x[1], x[0]))
    #选择最近的K个文本中出现次数最多的感情
    Kneighbors = [self.train_set[1][dis[z][0]] for z in range(self.K)]
    c = sorted(Counter(Kneighbors).items(), key=lambda x: (-x[1], x[0]))
    return c[0][0]
```

NB 分类：

1.计算 P(ei)，其中 self.labelnum 可以同时作为伯努利模型的分母部分

```
def calcPemotion(self):
    #计算P(ei) = 标签为ei的训练集文章数 / 训练集总文章数
    self.Pemotion = []
    NtrainDoc = len(self.train_set[0])
    for emotion in self.labelset:
        num = self.train_set[1].count(emotion)
        self.labelnum.append(num)
        self.Pemotion.append(num / NtrainDoc)
```

2.计算 P(xi | ei) 的分子部分 num，若是多项式模型，则 self.labeltotword 可为分母部分

```
def countword(self, ModelType):
    """
    #P(xi|ei)的分子部分 num
    多项式模型Multinomial: wordcnt[j][word] 表示情感ej下，单词word共出现了几次
    伯努利模型Bernoulli: wordcnt[j][word] 单词word共出现在了几篇情感为ej的文章中
    """
    for i in range(len(self.train_set[0])):
        label_index = self.labelset.index(self.train_set[1][i])
        if ModelType == "Multinomial":#多项式模型
            for word in self.train_set[0][i]:
                if word not in self.wordcnt[label_index]:
                    self.wordcnt[label_index][word] = 1
                else:
                    self.wordcnt[label_index][word] += 1
                self.labeltotword[label_index] += 1
        elif ModelType == "Bernoulli":
            for word in self.wordset:#伯努利模型
                if word in self.train_set[0][i]:
                    if word not in self.wordcnt[label_index]:
                        self.wordcnt[label_index][word] = 1
                    else:
                        self.wordcnt[label_index][word] += 1
```

3. 分类的预测概率计算部分

```
def predict(self, ModelType, predictType, alpha):
    self.countword(ModelType)
    totword, cnt = len(self.wordset), 0
    #选择需要对验证集还是测试集验证
    if predictType == "validate":
        testset = self.valid_set
    else:
        testset = self.test_set
        writer = csv.writer(open(self.writepath, "w", encoding="utf-8", newline=""), dialect='excel')
        writer.writerow(["textid", "label"])

    #计算需预测文本每一行testset[0][i]的各项情感的概率p
    for i in range(len(testset[0])):
        p = [1.0 for j in range(len(self.labelset))]
        for j in range(len(self.labelset)):
            for word in testset[0][i]:
                if word in self.wordset:
                    num = self.getwordnum(word, j)
                    if ModelType == "Multinomial": #多项式模型
                        p[j] *= ((num + alpha) / (self.labeltotword[j] + totword * alpha))
                    elif ModelType == "Bernoulli": #伯努利模型
                        p[j] *= (num + alpha) / (self.labelnum[j] + 2*alpha)
            p[j] *= self.Pemotion[j]
        p = list(np.array(p) / np.sum(np.array(p))) #对p归一化保证概率和为1
        label = self.labelset[p.index(max(p))]
        if predictType == "validate" and label == testset[1][i]:
            cnt += 1
        elif predictType != "validate":
            writer.writerow([i+1, label])
    print("Prediction has finished")
    if predictType == "validate":
        print("Acc", cnt / len(testset[0]))
        return  cnt / len(testset[0])
```

NB 回归：

1.得到 TF 矩阵

```
def getTF(self, Matrix):
    num, den = [], []
    for i in range(len(Matrix)):
        num.append([])
        den.append(0)
        for word in self.wordset:
            cnt = Matrix[i].count(word)
            num[i].append(Matrix[i].count(word))
            den[i] += cnt
    return [num, den]
```

```
[self.trainNUM, self.trainDEN] = self. getTF(self.train_set[0])
```

2.计算P(word|e，train[k])

```
def getP(self, word, traindoc, k, alpha):
    index = self.wordset.index(word)
    a= self.trainNUM[k][index] + alpha
    b= self.trainDEN[k] + self.totword * alpha
    return a/b
```

3.预测

```
def predict(self, predictType, alpha):
    #选择要预测的集合是验证集还是测试集
    if predictType == "validate":
        testmatrix = self.valid_set
    else:
        testmatrix = self.test_set
    writer = csv.writer(open(self.writepath, "w", encoding="utf-8", newline=""), dialect='excel')
    writer.writerow(["textid"] + self.labelset)
    #计算概率
    for i in range(len(testmatrix[0])):
        p = [0 for j in range(len(self.labelset))]
        for j in range(len(self.labelset)):
            for k in range(len(self.train_set[0])):
                temp = 1.0
                for word in set(testmatrix[0][i]):
                    if word not in self.wordset:
                        continue
                    #计算P(word|e, train[k])
                    temp *= self.getP(word, self.train_set[0][k], k, alpha)
                #计算temp *=  p(ej|train[k])
                temp *= self.train_set[1][k][j]
                p[j] += temp #累加概率
        p = list(np.array(p) / np.sum(np.array(p)))#归一化
        writer.writerow([i+1] + p)
        print("text ", i, "has finished")
    print("Prediction has finished")
```

4. 创新点&优化（如果有）

KNN：

1.Z – Score 标准化的时候为了防止除 0，在分母加上了一个极小的数 1^(-10)

2.使用了 TFIDF 矩阵 + 余弦距离优化结果

3.如果出现了跟测试文本完全一样的训练文本，则直接使用它的概率。

NB:

1.分类实现了伯努利模型

计算伯努利模型P(word|ei)的分子：

```
elif ModelType == "Bernoulli":
    for word in self.wordset:#伯努利模型
        if word in self.train_set[0][i]:
            if word not in self.wordcnt[label_index]:
                self.wordcnt[label_index][word] = 1
            else:
                self.wordcnt[label_index][word] += 1
```

计算伯努利模型的P(word|ei)

```
elif ModelType == "Bernoulli": #伯努利模型
    p[j] *= (num + alpha) / (self.labelnum[j] + 2*alpha)
```

由于计算过程没有和多项式模型完全分开，在前面关键代码展示中已经全

部展示过，这里就不再重复展示。

2. 通过调整 alpha 值来提供准确率（回归的话是相关度）

三、 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

KNN 分类：

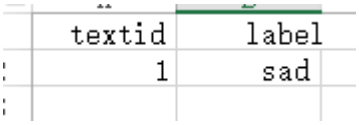
set:

Words	label
Good thanks	joy
No impressive thanks	Sad
Impressive good	Joy
No thanks	?

处理成 onehot 矩阵:

ID	Good	Thanks	No	Impressive	Lable
train1	1	1	0	0	Joy
train2	0	1	1	1	Sad
train3	1	0	0	1	Joy
test1	0	1	1	0	?

取 K=1，欧几里得距离跑出的结果为:



textid	label
1	sad

验证结果:

d(test1, train1) = $\sqrt{2}$

d(test1, train2) = 1

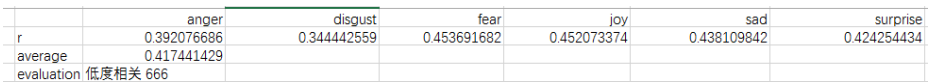
d(test1, train3) = 2

则应取第二个文本的标签，即 sad，所以程序得出的结果准确

KNN 回归：

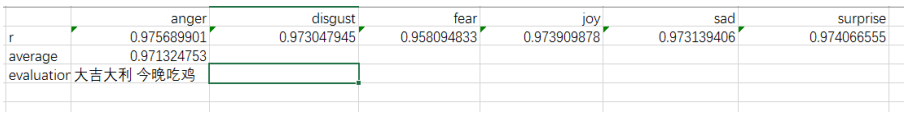
跑的是实验课给的数据

使用所有优化，Z-SCORE 标准化 + TFIDF + 余弦距离 + 特判完全一样的文本



	anger	disgust	fear	joy	sad	surprise
r	0.392076686	0.344442559	0.453691682	0.452073374	0.438109842	0.424254434
average	0.417441429					
evaluation	低度相关 666					

与其他同学对拍了答案的相关度:



	anger	disgust	fear	joy	sad	surprise
r	0.975689901	0.973047945	0.958094833	0.973909878	0.973139406	0.974066555
average	0.971324753					
evaluation	大吉大利 今晚吃鸡					

相关度达到了 97，程序基本正确。

NB 分类：

使用 KNN 分类中的小数据集，取 alpha=1，多项式模型得出结果:

textid	label
1	sad

验证答案:

P(joy) = 2/3

P(sad) =1/3

P(test1 | joy) = (2/11) * (1/11) *(2/3)= 0.011

P(test1 | sad) = (1/5) * (1/5) *(1/3)= 0.013

则 P(test1 | joy) < P(test1 | sad)

结果为 sad，正确

NB 回归：

取 alpha=0.032 跑实验课给的数据集的结果如下：

	anger	disgust	fear	joy	sad	surprise
r	0.325525346	0.225269745	0.377432827	0.419331199	0.400554621	0.404770461
average	0.358814033					
evaluation	低度相关 666					

与其他同学对拍了答案的相关度:

	I	J	K	L	M	N	O
		anger	disgust	fear	joy	sad	surprise
	r	0.98221175	0.983540638	0.988360886	0.99104013	0.98348144	0.979442784
	average	0.984679605					
	evaluation	大吉大利 今晚吃鸡					

2. 评测指标展示即分析（如果实验题目有特殊要求，否则使用准确率）

KNN 分类：

1. 最普通的 onethot + 曼哈顿距离 + 不标准化：

```
MatrixType:  onehot
standard:  original
disType:  Manhattan
K =  5 Accuracy =  0.34726688102893893
K =  6 Accuracy =  0.3633440514469453
K =  7 Accuracy =  0.3440514469453376
K =  8 Accuracy =  0.3215434083601286
K =  9 Accuracy =  0.3665594855305466
K =  10 Accuracy =  0.3762057877813505
K =  11 Accuracy =  0.3987138263665595
K =  12 Accuracy =  0.3890675241157556
K =  13 Accuracy =  0.3954983922829582
K =  14 Accuracy =  0.3890675241157556
K =  15 Accuracy =  0.39228295819935693
K =  16 Accuracy =  0.40514469453376206
K =  17 Accuracy =  0.3987138263665595
K =  18 Accuracy =  0.3987138263665595
K =  19 Accuracy =  0.3729903536977492
```

2. 使用所有优化，Z-SCORE 标准化 + TFIDF + 余弦距离

```
问题    输出    调试控制台    终端
MatrixType:  TFIDF
standard:  normalization
disType:  Cosine
K =  5 Accuracy =  0.40192926045016075
K =  6 Accuracy =  0.40192926045016075
K =  7 Accuracy =  0.41479099678456594
K =  8 Accuracy =  0.4405144694533762
K =  9 Accuracy =  0.4340836012861736
K =  10 Accuracy =  0.4405144694533762
K =  11 Accuracy =  0.4437299035369775
K =  12 Accuracy =  0.43729903536977494
K =  13 Accuracy =  0.42765273311897106
K =  14 Accuracy =  0.43729903536977494
K =  15 Accuracy =  0.41479099678456594
K =  16 Accuracy =  0.40514469453376206
K =  17 Accuracy =  0.39228295819935693
K =  18 Accuracy =  0.3987138263665595
K =  19 Accuracy =  0.3633440514469453
```

可以发现普通做法，准确率最高只有 0.405，而优化了之后，当 K=11 时，准确率达 0.4437。说明优化的确可以帮助提高预测准确率

KNN 回归：

使用分类得到的最优 K=11 来预测

1. 最普通的 onethot + 曼哈顿距离 + 不标准化：

	anger	disgust	fear	joy	sad	surprise
r	0.236589887	0.196361121	0.307975139	0.306026073	0.298411242	0.280379446
average	0.270957151					
evaluation	极弱相关 加油哦小辣鸡					

可以发现相关度非常低，只有 0.27，其中对 disgust 的预测最不准确

2. 使用所有优化，Z-SCORE 标准化 + TFIDF + 余弦距离 + 特判完全一样的文本

	anger	disgust	fear	joy	sad	surprise
r	0.392076686	0.344442559	0.453691682	0.452073374	0.438109842	0.424254434
average	0.417441429					
evaluation	低度相关 666					

可以看到相关度提高了很多，说明优化很有作用。

NB 分类：

NB 主要调节的是 alpha 的值，由于我跑了(0,1] 之间，间隔 0.001 的所有值去当 alpha，结果太多，下面节选一些 alpha 值的结果作为展示

多项式模型:

Alpha	Accuracy
0.001	0.42765273311897106
0.055	0.4340836012861736
0.16	0.4437299035369775
0.2	0.4405144694533762
0.35	0.45016077170418006
0.42	0.4662379421221865
0.55	0.47266881028938906

0.6	0.4694533762057878
0.7	0.45980707395498394
0.8	0.4565916398713826
0.9	0.4565916398713826
1.0	0.4565916398713826

由表可知，当 alpha = 0.55 时，准确率最高，为 0.4727

伯努利模型:

Alpha	Accuracy
0.01	0.4115755627009646
0.05	0.40192926045016075
0.1	0.38263665594855306
0.2	0.3311897106109325
0.3	0.2861736334405145
0.4	0.2282958199356913
0.5	0.2090032154340836
0.6	0.17684887459807075
0.7	0.14469453376205788
0.8	0.13504823151125403
0.9	0.13504823151125403
1.0	0.12861736334405144

由数据可以看出，伯努利的预测效果远不及多项式模型，多项式模型比较稳定，伯努利模型的正确率与 alpha 基本成负相关。

NB 回归：

Alpha = 0.032

	anger	disgust	fear	joy	sad	surprise
r	0.325525346	0.225269745	0.377432827	0.419331199	0.400554621	0.404770461
average	0.358814033					
evaluation	低度相关 666					

Alpha = 1

	anger	disgust	fear	joy	sad	surprise
r	0.266904431	0.159685317	0.288178138	0.324663282	0.315230934	0.290193424
average	0.274142588					
evaluation	极弱相关 加油哦小辣鸡					

四、 思考题

KNN：

1. 回归问题中，为什么要将距离的倒数作为权重？

答：KNN 回归问题是将 K 个邻居的平均属性赋值给当前测试样本，但这样不

够准确，应该根据距离的远近，让邻居对这个样本的影响程度有所不同，距离越近的影响越大，因此想到根据距离来加权，取倒数就满足距离越小，权重越大的要求。

2. 同一测试样本的各个情感概率总和应该为 1，如何处理？

答：只要在最后对情感概率归一化处理，同比放大即可。

3. 在矩阵稀疏程度不同的时候，曼哈顿距离和欧式距离的表现有什么区别，为什么？

答：我个人觉得，没有什么区别。我认为这两者的表现的区别不在于矩阵的稀疏，而在于向量特征值的差别。假设一个测试样本的特征向量跟很多训练样本的特征向量的欧式距离相同，可能会没办法好好得出正确的类别，不过这个时候虽然欧式距离相同，当曼哈顿距离不一定相同，也许使用曼哈顿距离会更好。而曼哈顿距离如果向量中存在一个特征值相差较大的话，可能会掩盖其他特征值的变化特征。

硬要说矩阵稀疏程度不同两者表现有什么区别的话，假如使用了三元组表示矩阵，随着矩阵越来越密集，欧式距离的计算时间会更大。

NB：

1. 分类问题中，伯努利模型和多项式模型有什么优缺点？

答：伯努利模型是文档型的模型，它的特征向量是单词有没有出现过，是一个只含（0,1）两种取值的向量，是以单词出现的文档数占总文档数的概率去做预测，是一种从全局特征考虑的预测方式，实现起来比较简单，缺点就是由于考虑的是全局特征，从而忽略了局部特征的影响，比如同样一个词，在一篇感情为 ei 的文章中出现了很多次，但是在一篇感情为 ej 的文章中只出现了 1 次的话，那么我们直观上应该将这个词归为 ei 这类，然后在伯努利模型中，这个词是感情 ei 或 ej 的概率都是 1/2，这样的预测就不够精确。

多项式模型是单词型的模型，它的特征向量是词频，也就是说它将单词出现的频率考虑了进去，这样就可以解决上述伯努利模型中的忽略局部特征影响的问题。缺点是当特征是连续变量时，使用多项式模型会造成后验概率为 0 的情况，即时使用了平滑处理也不能很好地反映真实情况。

2. 如果测试集中出现了一个之前全词典中没有出现过的词该如何解决？

答：这种情况下我选择忽略这个词,当成测试集中不存在它。原因是由于全词典中没有这个词，那么就不知道跟它有关的各种概率，无法去参与后验概率计算，既然如此，不如选择舍弃。