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	**M1**	专业（方向）	**移动信息工程（互联网方向）**
学号	**15352102**	姓名	**韩硕轩**

一、 实验题目

K 近邻与朴素贝叶斯的分类与回归

二、 实验内容

1. 算法原理

（1） K 近邻的分类。原理是对于每一个验证文本或者测试文本，找出训练集中与其词向量距离最近的 k 个文本，然后通过某种决策方式确定下来新的文本的标签。词向量可以是 onehot 矩阵，可以是 TF 矩阵，也可以是 TF-IDF 矩阵。距离的测量可以是曼哈顿距离，可以是欧氏距离，也可以是两个词向量的余弦相似度。决策的方式一般是取众数。K 值通过验证集确定，在验证集上表现最好的 k 将会被应用到测试集答案的计算中。

（2） K 近邻的回归。原理和上面的分类相似，由词向量算距离，取出前 k 个相近的训练文本。接下来给距离取倒数作为权重，对 k 个训练文本进行加权算出新文本对应标签的可能性。也就是按照下图公式算，这是 k 为 3，标签为 happy 的情况：

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

（3） 朴素贝叶斯的分类。采用多项式模型，每个验证和测试文本作为一个词袋，整个词袋的先验概率是其中每个词的先验概率之积。而每个词的先验概率等于这个词在对应标签的文本中出现的次数除以这个标签的总词数，二者均是可重复的。然后该标签的后验概率正比于词袋的先验概率乘似然度，也就是该标签在所有标签中占的比重。最后取出可能性最大的标签即为分类的答案。加入拉普拉斯平滑之后，每个单词的先验概率在计算的时候，分子加 1，分母加所有文本中不重复的单词总数。

（4） 朴素贝叶斯的回归。对于一个标签，似然度需要枚举它和每一个训练文本的组合，每次枚举还需要再用一个子循环乘上词袋中每一个单词的先验概率，然后将每个训练文本算出来的值相加，得出该标签概率的标准化常量倍。最后算出所有标签之后，对这个新文本的所有标签进行归一化，即为回归的答案。

2. 伪代码

（1） K 近邻的分类：

```
While (训练集没结束)
{
    Getline(训练文本);
    While (分词没结束)
    {
        统计训练的单词或者记录标签;
        分词;
    }
}
While (验证集没结束)
{
    Getline(验证文本);
    While (分词没结束)
    {
        统计验证的单词或者记录标签;
        分词;
    }
}
根据统计结果算出 onehot, TF, TF-IDF;
For k from 1 → sqrt(训练集文本个数)
    For I in 验证集
        算出第 i 个验证文本和每个训练文本的距离;
        取出前 k 个;
        算出众数，取标签;
        判断是否和读入的标签相同，并统计;
    If (这个 k 正确率更高) 记录这个 k;
读入测试集;
重复在验证集上的计算步骤，将答案写入文本;
```

（2） K 近邻的回归：

```
初始化将文本改成一行一个单词，便于读入;
读入和统计与上一个相同;
根据统计结果算出 onehot, TF, TF-IDF;
For k from 1 → sqrt(训练集文本个数)
    For I in 验证集
        For j in {emotions}
            算出第 i 个验证文本和每个训练文本的距离;
            取出前 k 个;
            根据公式算出这个标签的可能性;
        归一化这些标签;
    输出所有的验证集的答案，查看相关性;
取出最优的 k;
```

读入测试集，重复计算步骤，用 k 算出答案;

（3） 朴素贝叶斯的分类：

读入部分和 KNN 分类的读入相同;

统计$nw_{ei}(x_k), nw_{ei}, V_{ei}$;

```
While (验证集没结束)
{
    Getline(验证文本);
    While (分词没结束)
    {
        分词;
        根据nw_ei(x_k), nw_ei, V_ei更新每种标签的概率;
    }
    找出概率最大的标签作为该验证文本答案;
}
```

统计正确率;

读入测试集，算出答案写入文本;

（4） 朴素贝叶斯的回归：

初始化和读入部分和 KNN 回归的相同;

分词过程和 KNN 回归的相同;

读入验证集部分和 KNN 回归的相同;

统计 onehot 矩阵;

统计加入了拉普拉斯平滑的 TF 矩阵，分子加 1，分母加不重复的单词总数;

```
For I in  验证集
    For j in {emotions}
        For k in  训练集
            Temp ← 第 k 个训练文本的情感 j 的概率
            For l in  所有单词
                If (验证集 i 出现了单词 l)
                    Temp ← temp*TF[k][l]
        第 i 个验证文本的第 j 个情感的概率  += temp;
    输出六个情感的概率;
```

读入测试集，做同样的处理，输出答案，写入文本;

3. 关键代码截图（带注释）

（1） K 近邻的分类：

变量的定义部分，作用见注释。

```
struct dist {
    double dis;
    int trnum;
};

struct emotion {
    string es;
    int count;
};

int num, txt1, txt2, txt3;    //分别是不同的单词总数，训练集、验证集和测试集句子数
int ori[1500][15000];     //原始的单词统计矩阵
                          //其中第0行表示每个单词在多少个文本中出现过（算idf用），第0列表示每句话有多少个单词（算tf用）
int OH[1500][15000];            //onehot矩阵
double TF[1500][15000], TFIDF[1500][15000];    //TF矩阵和TFIDF矩阵
string words[15000];       //按照出现顺序记录每一个单词
string label[1500];     //记录训练集中每个文本的label和验证集的答案
int k, cnt;
emotion em[6];
dist d[1500];                                 //记录距离
```

读入训练集并分词。

```
while (!trainin.eof())        //一行一行读入训练集
{
    trainin.getline(s, 500);
    txt1++;
    const char *d = " ";
    char *p;
    p = strtok(s, d);
    while (p)                                 //这个while循环用来分词
    {
        string ss = p;
        int comma = ss.find(",", 0);
        if (comma < string::npos)
        {
            label[txt1] = ss.substr(comma + 1, ss.length() - comma - 1);
            p[comma] = '\0';
        }
```

对分出来的单词做处理。

```
int pos = search(p);          //判断是新词还是旧词并做不同处理
ori[txt1][0]++;
if (pos > 0)     //旧单词直接找到位置，统计ori
{
    ori[txt1][pos]++;
    if (ori[txt1][pos] == 1) ori[0][pos]++;
}
else     //新单词则新开一个位置，统计ori并录入words
{
    num++;
    ori[txt1][num]++;
    ori[0][num]++;
    words[num] = p;
}
p = strtok(NULL, d);        //分出下一个单词，81行和58行一起分词
```

算出三种矩阵。

```
for (int i = 1; i <= txt1 + txt2; i++)              //这个双重循环算出onehot矩阵
    for (int j = 1; j <= num; j++)
        if (ori[i][j] > 0) OH[i][j] = 1;
        else OH[i][j] = 0;
for (int i = 1; i <= txt1 + txt2; i++)              //这个双重循环算出TF矩阵
    for (int j = 1; j <= num; j++)
        TF[i][j] = ori[i][j] * 1.0 / ori[i][0];

for (int i = 1; i <= txt1 + txt2; i++)              //这个双重循环算出TFIDF矩阵
    for (int j = 1; j <= num; j++)
        TFIDF[i][j] = TF[i][j] * log((txt1 + txt2) * 1.0 / (ori[0][j] + 1)) / log(2);
```

循环枚举 k，得出正确率最高的 k。

```
k = 0;
int maxk = 0, maxc = 0;
em[0].es = "anger";
em[1].es = "disgust";
em[2].es = "fear";
em[3].es = "joy";
em[4].es = "sad";
em[5].es = "surprise";
while (k <= int(sqrt(txt1))) { ... }
k = maxk;
```

算出距离。（这里以 onehot 矩阵加曼哈顿距离为例）

```
for (int i = 1; i <= txt2; i++)
{
    for (int j = 1; j <= txt1; j++)
        d[j].dis = 0;
    for (int j = 1; j <= txt1; j++) //算出训练集和验证集每一对向量之间的距离
    {
        for (int 1 = 1; 1 <= num; 1++)
        {
            d[j].dis += abs(OH[i + txt1][1] - OH[j][1]);
        }
        d[j].trnum = j;
    }
```

算出每一个 k 值下对应验证文本的答案。

```
sort(d + 1, d + txt1, cmp); //对所有的距离排序
for (int j = 0; j < 6; j++)
    em[j].count = 0;
for (int j = 1; j <= k; j++)  //取出前k近的训练文本
    if (label[d[j].trnum] == "anger") em[0].count++;
    else if (label[d[j].trnum] == "disgust") em[1].count++;
    else if (label[d[j].trnum] == "fear") em[2].count++;
    else if (label[d[j].trnum] == "joy") em[3].count++;
    else if (label[d[j].trnum] == "sad") em[4].count++;
    else if (label[d[j].trnum] == "surprise") em[5].count++;
    int maxx = 0;
    string lb;
    for (int j = 0; j < 6; j++)
    {
        if (em[j].count >= maxx)
        {
            maxx = em[j].count;  //找到众数
            lb = em[j].es;   //记下标签
        }
    }
    if (lb == label[i]) cnt++;  //最终出来的这个标签就是答案
    res << lb << "," << label[i] << endl;
```

统计正确率，保存最大的 k。

```
    cout << k << " " << cnt*1.0 / txt2 << endl; //统计正确率
    if (cnt > maxc)
    {
        maxc = cnt;
        maxk = k;
    }
}
k = maxk;  //让k取最大的正确率的k
```

接下去的测试集的读入和处理都是上面已经截图部分代码复制过去改变量名得到的，所以不再赘述。下面的 K 近邻回归和朴素贝叶斯分类回归的读入处理也同理不再重复截图。

（2） K 近邻的回归：

我首先对输入文本进行了预处理，改成一行一个单词，便于读入。这样做会产生大量写入文本的时间，造成时间的浪费，优点可能只是代码写起来方便。下次我会想办法改进。

```
trainin.getline(s, 500);
while (!trainin.eof())
{
    trainin.getline(s, 500);
    for (int j = 0; j<strlen(s); j++)
        if (s[j] == ' ' || s[j] == ',') trainout << endl;
        else trainout << s[j];
        trainout << endl;
}
validationin.getline(s, 500);
while (!validationin.eof())
{
    validationin.getline(s, 500);
    for (int j = 0; j<strlen(s); j++)
        if (s[j] == ' ' || s[j] == ',') validationout << endl;
        else validationout << s[j];
        validationout << endl;
}
```

接下来是算距离，算结果并归一化的过程。（多次测试发现 k 为 txt1 时候相关度最高）

```
for (int i = 1; i <= txt2; i++)
{
    for (int j = 1; j <= txt1; j++)
        d[j].dis = 0;
    for (int j = 1; j <= txt1; j++)
    {
        for (int l = 1; l <= num; l++)  //算出距离
        {
            d[j].dis += abs(TFIDF[i + txt1][l] - TFIDF[j][l]);
        }
        d[j].dis2 = 1.0 / d[j].dis; //取距离的倒数
    }
    double sum = 0;
    for (int j = 1; j <= txt1; j++)
    {
        for (int l = 0; l<6; l++)  //根据距离的倒数乘上概率得到答案
        {
            sum += d[j].dis2*emotion[j][l];
            ans[i][l] += d[j].dis2*emotion[j][l];
        }
    }
    for (int j = 0; j<6; j++)
    {
        ans[i][j] = ans[i][j] * 1.0 / sum;  //归一化
        res << ans[i][j] << ",";
    }
```

同样，最后的测试集处理不再截图。

（3） 朴素贝叶斯的分类：

在写 NB 的时候做了一点改进，就是不再每次用 if 语句判断，而是放到一个函数里面。

```
int emorder(string emo)
{
    if (emo == "anger") return 0;
    else if (emo == "disgust") return 1;
    else if (emo == "fear") return 2;
    else if (emo == "joy") return 3;
    else if (emo == "sad") return 4;
    else if (emo == "surprise") return 5;
}
```

Emotion 的定义变了。

```
struct emotion {
    string es;
    int ne, nw, ve;          //文本个数、单词总数和不重复单词总数
    string ewords[10000];
};
```

还新增了 new 便于统计。

```
int nwe[6][15000];          //每个单词在不同标签中出现次数
```

这是分词时候的统计。

```
while (p)                                   //这个while循环用来分词
{
    int pos = search(p);                    //判断是新词还是旧词并做不同处理
    em[emod].nw++;
    if (pos>0)      //这是一个旧单词
    {
        nwe[emod][pos]++;   //原位置加一即可
    }
    else
    {
        num++;      //新单词要新开位置
        nwe[emod][num]++;
        words[num] = p;
    }
    if (search2(p)<0)
    {                           //标签中的新单词要记录下来
        em[emod].ewords[em[emod].ve] = p;
        em[emod].ve++;
    }
    p = strtok(NULL, d);                    //分出下一个单词，81行和58行一起分词
}
```

验证集分词的同时就做统计。

```
while (p)                                   //这个while循环用来分词
{
    string ss = p;
    int comma = ss.find(",", 0);
    if (comma < string::npos)
    {
        label[txt2] = ss.substr(comma + 1, ss.length() - comma - 1);
        p[comma] = '\0';
    }
    for (emod = 0; emod<6; emod++)
    {  //按照公式统计每个单词在各个标签下的先验概率
        int pos = search(p);
        if (pos<0) pos = 0;
        pe[emod] *= ((nwe[emod][pos] + 1)*1.0 / (em[emod].nw + num2));
    }
    p = strtok(NULL, d);
}
```

找出最大值并输出，同时统计正确率。

```
    string lb;
    double maxx = 0;
    for (int i = 0; i<6; i++)
    {
        if (pe[i]>maxx)
        {
            maxx = pe[i];
            lb = em[i].es;
        }
    }
    if (lb == label[txt2]) cnt++;
}
txt2--;
cout << cnt*1.0 / txt2 << endl;
```

（4） 朴素贝叶斯的回归：

一开始同样做了数据集的预处理，便于读入，见（2）。

读入的分词和统计过程与（3）一样。

由于统计需要，我又统计了 ori 矩阵。

```
trainin2.getline(s, 500);
if (isdigit(s[0]))
{
    emotion[txt1][0] = atof(s);
    for (int j = 1; j<6; j++)
    {
        trainin2.getline(s, 500);
        emotion[txt1][j] = atof(s);
    }
    txt1++;
}
else
{
    int pos = search(s);                    //判断是新词还是旧词并做不同处理
    ori[txt1][0]++;
    if (pos > 0)
    {
        ori[txt1][pos]++;
        if (ori[txt1][pos] == 1) ori[0][pos]++;
    }
    else
    {
        num++;
        ori[txt1][num]++;    //统计ori
        ori[0][num]++;
        words[num] = s;
    }
}
```

算 TF 矩阵。

```
for (int i = 1; i <= txt1 + txt2; i++)
    for (int j = 1; j <= num; j++)  //算出onehot
        if (ori[i][j]>0) OH[i][j] = 1;
        else OH[i][j] = 0;
for (int i = 1; i <= txt1 + txt2; i++)              //这个双重循环算出TF矩阵
{
    for (int j = 1; j <= num; j++)
    {
        TF[i][j] = (ori[i][j] + 1) * 1.0 / (ori[i][0] + num2);
    }   //加入拉普拉斯平滑
}
```

四重循环算出答案并输出。

```
memset(ans, 0, sizeof(ans));
for (int i = 1; i <= txt2; i++) //验证集
{
    double pe[6], sum = 0;
    for (int j = 0; j<6; j++) //情感
    {
        pe[j] = 0;
        for (int k = 1; k <= num2; k++) //训练集
        {
            double temp = emotion[k][j];
            for (int l = 1; l <= num; l++)  //单词
                if (ori[i + txt1][l]>0) temp *= TF[k][l];
            pe[j] += temp;
        }
        sum += pe[j];
    }
    for (int j = 0; j<6; j++)
    {
        pe[j] = pe[j] / sum;
        res << pe[j] << ",";
    }
    res << "line" << endl;
}
```

测试集也是同理，不再截图。

4. 创新点&优化（如果有）

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）

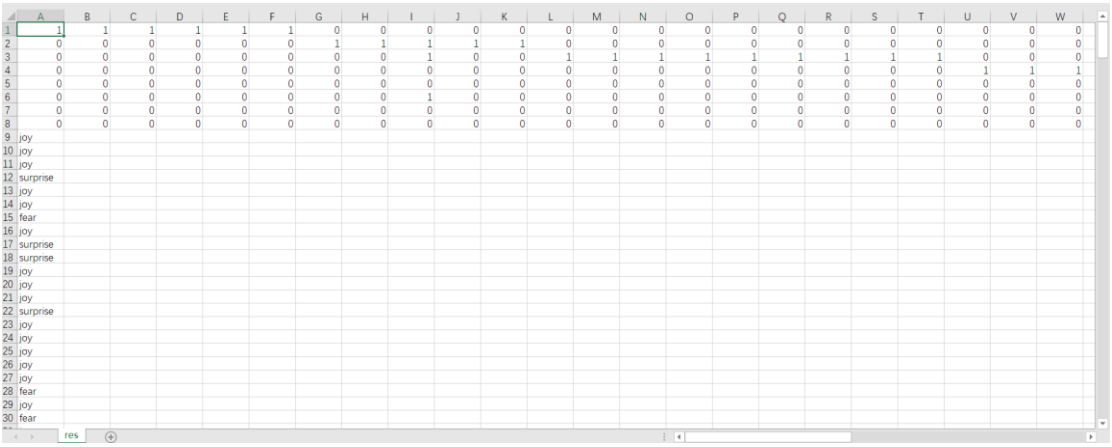
在分类中，我先截取 train 的前几行，创建了一个小的训练数据集：

	A	B	C
1	Words (split by space)	label	
2	europe retain trophy with big win	joy	
3	senate votes to revoke pensions	sad	
4	the amounts you have to pay for a bomb scare	fear	
5	pair of satellites will document sun in d	joy	

同时验证集也改成小数据：

	A	B
1	Words (split by space)	label
2	marijuana helps ease hiv nerve pai	surprise
3	thousands line up to get late flu sh	fear
4	king county offering some free flu	joy
5	rig threat cargo ship towed ashore	fear

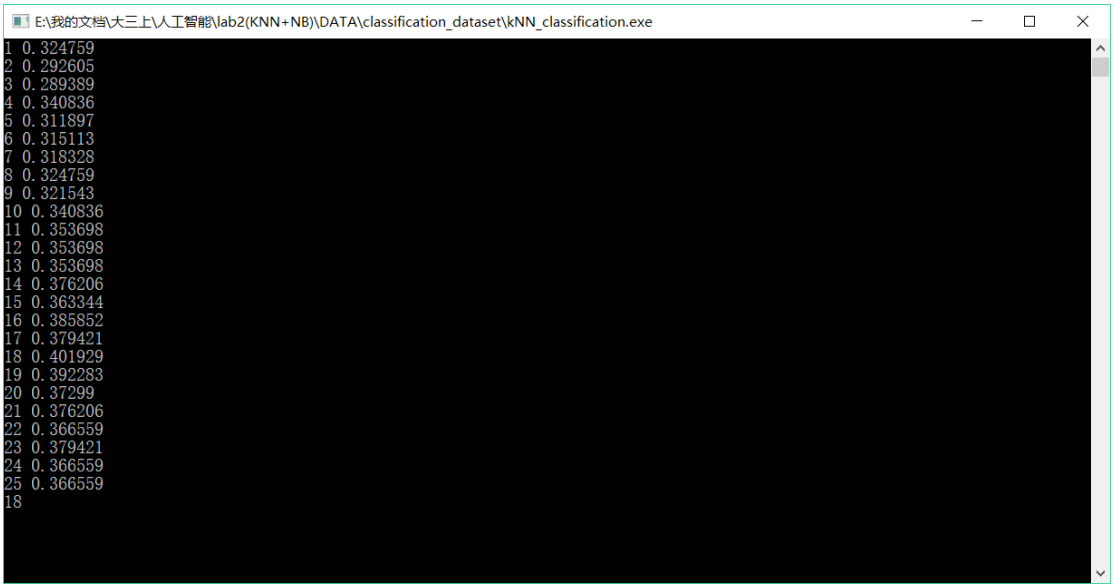
然后输出这个：



通过手动验证，得出代码的正确性。

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）

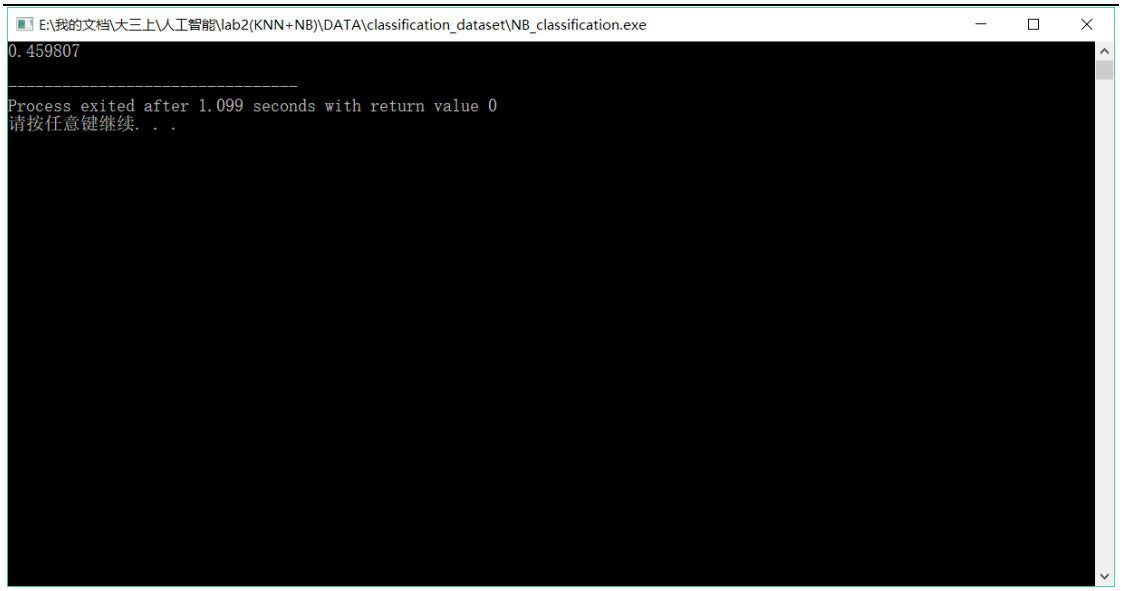
KNN 的分类：



KNN 的回归：

	anger	disgust	fear	joy	sad	surprise
r	0.291619761	0.271205233	0.400769932	0.415193418	0.387763011	0.391805056
average	0.359726069					
evaluation	低度相关 666					

NB 的分类：



NB 的回归：（这里严重怀疑自己写错了但是没找到 bug）

	anger	disgust	fear	joy	sad	surprise
r	0.67235119	0.61868284	0.649008405	0.662220696	0.693658665	0.6915505
average	0.664578716					
evaluation	中度相关 大神！					

四、 思考题

1.为什么相似度加权的时候权重是距离的倒数呢？

答：因为距离越近，数值越小，采用倒数，权重才会越大。而距离越近的测试文本需要占有越大的比重，刚好就符合要求，所以采用距离倒数。

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：归一化。先算出所有情感的和，然后每一个情感的可能性除以这个和，最后得到的所有情感的和为 1。

3.在矩阵稀疏程度不同的时候，曼哈顿距离和欧氏距离表现有什么区别，为什么？

答：个人思考的是，曼哈顿距离在矩阵非常稀疏的时候适用，而欧氏距离在矩阵稀疏程度不那么大的时候适用。因为曼哈顿距离是一个整数，而欧氏距离大概率是一个无理数，不同向量之间曼哈顿距离相同的概率比欧氏距离相同的大，所以欧氏距离更容易区分出距离的远近，也就是更有区分度。但是在矩阵特别稀疏的时候，用欧式距离很容易算出很小的数值，而曼哈顿距离则表现得更直观更有区分度一些。

4.伯努利模型和多项式模型分别有什么优缺点？

答：伯努利的优点是以文件为粒度，可以较好的排除噪声的干扰，使结果更稳定。多项式模型优点在于以单词为粒度，这样在噪声较小的情况下结果更精确。

5.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：用拉普拉斯平滑可以避免出现因为一个词导致整个词袋概率为 0 的情况。方法是多项式模型中给每个单词后验概率分子加一，分母加上标签 ei 中不重复的单词数。而回归模型中方法是每个单词后验概率分子加一，分母加上不重复单词总数。

|---------- 如有优化，重复 1，2 步的展示，分析优化后结果 ----------|

PS：可以自己设计报告模板，但是内容必须包括上述的几个部分，不需要写实验感想