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	**15M1**	专业（方向）	**移动互联网**
学号	**15352048**	姓名	**陈潇**

一、 实验题目

文本数据集的简单处理

二、 实验内容

1. 算法原理

1） 有监督的机器学习是通过编写算法，让算法对带有标签的训练数据进行存储、计算，以此使用训练数据将训练模型训练到一定程度后，再使用训练好的模型来预测不带标签的数据的标签。本次的实验内容就是编写一个有监督的机器学习算法。

2） KNN（K 近邻）算法是有监督的机器学习模型的一种。其分类算法原理为计算待测试文本数据与所有训练数据之间的距离（可以取欧氏距离、曼哈顿距离等）存储到集合{A}中，再取{A}中距离最小的前 k 个训练数据的标签作为【有效投票】，最终投票（众数）得出待测试文本数据的【标签】，在投票【平票】的情况下，可以随机选取标签；在前 k 个训练数据出现距离相同的情况下，可以选取最先出现的训练数据。

3） KNN 的回归算法原理类似于分类算法，不同点在于回归需要计算所有标签下的结果。

4） NB（朴素贝叶斯）算法是有监督的机器学习模型的一种。其分类算法原理为根据【标签】去寻找对应训练数据中出现【待测试文本数据】的频率，乘以训练数据的【先验概率】得到该【标签】前提下的预测结果。最终计算出所有的预测结果后取概率最大的作为待测试文本数据的【标签】。

5） NB 的回归算法不再根据标签，而是对于每一个训练数据的【标签概率】，都要去寻找待测试文本数据的所有非零对应位置的 TF 矩阵的数据，作为频率相乘，再与【标签概率】相乘，最后再求这些结果的和，得到待测试文本的【标签概率】。

6） 拉普拉斯平滑是解决【待测试文本】的数据没有出现在【训练数据】中的情况，避免出现概率计算的过程中分子为 0 或分母为 0 的情况。

2. 伪代码

```
string KNN_classification(int k)
{
    打开 TF 矩阵文件；
    while(按行读 TF 矩阵入到字符串 s)
    {
        处理字符串 s，把 s 转化为 double 转存到 vector 数组 singleRow 中;
        For(对 singleRow 中的每一项 i)
          {
              计算欧氏距离
              d += (singleRow[i] – testData[i])^2
          };
        For(对于没有出现在单词表中的单词 i)
          {
              d += testData[i] ^ 2
          }
        对 d 取模。
        将 d 加入到欧式距离集合 E;
    }
    对集合 E 进行排序。
    取前 k 个进行有效投票（众数）选择；
    得到标签，返回结果
}
```

```
Double* KNN_regression(int k)
{
    打开 TF 矩阵文件；
    while(按行读 TF 矩阵入到字符串 s)
    {
        处理字符串 s，把 s 转化为 double 转存到 vector 数组 singleRow 中;
        For(对 singleRow 中的每一项 i)
          {
              计算欧氏距离
              d += (singleRow[i] – testData[i])^2
          };
        For(对于没有出现在单词表中的单词 i)
          {
              d += testData[i] ^ 2
          }
        对 d 取模。
        将 d 加入到欧式距离集合 E;
    }
    对集合 E 进行排序
    For(循环 k 次，i)
    {
        计算 E[i]对应在训练文本里的数据（情感概率） / E[i]的欧氏距离;
    }
    存储结果;
    返回结果（数组）;
}
```

```
string NB_classification()
{
    打开输入文本;
    for(对待测试数据集合 testData 中的每一句 s)
    {
        If(testData[i] == 1)
        {
            循环 6 次寻找所有情感, i1;
            For(对于所有训练数据)
            {
                If(当前训练数据的情感标签==第 i1 种情感)
                    If(训练数据[i] == 1)  当前情感 i1 ++;
            }
            最终情感结果 E.push_back(拉普拉斯平滑(当前情感 i1/i1 标签下训练数据中的 1
     个数))
        }
    }
    循环 6 次寻找最终情感结果 E 中概率最大者，记录标签
    返回标签结果;
}
```

```
Double* NB_regression()
{
    打开 TF 矩阵 eachRow;
    for(循环 6 次寻找所有情感标签的概率,i1)
    {
        for(对于每一个训练数据,j)
        {
            For(对于训练数据[j])
            {
                If(待测试数据 testData[i] != 0 )
                    情感概率 temp *=  拉普拉斯平滑（eachRow[i]）
            }
            Temp *= eachRow 中 i1 情感的先验概率
            最终情感结果 E.push_back(temp)
        }
    }
    对 E 进行归一化处理
    返回数组 E;
}
```

其余的代码部分为对训练文本进行切词处理、生成 OneHot 矩阵、生成 TF 矩阵、对待测试文本进行切词处理、循环生成 KNN、NB 的结果并验证等，这些代码与上一次实验的大部分内容类似，不再赘述。

3. 关键代码截图（带注释）

为了便于 TA 阅读，先对以下所有变量进行解释：

（vector string）ArticleSet=所有训练文本，不包含情感标签

（vector string）DataSet=单词表，不重复

（vector double）singleRow=矩阵处理后的每一个训练文本的数据（称为训练数据）

（map pair<int, double>）EuclideanDistance=欧氏距离集合，对应<行号,距离>

（vector double）testData=当前待测试文本的数据（待测试数据，只有一行）

3.1 KNN 部分

```
    fstream tf("TF.txt");
    double dtemp;
    //对于每一行训练数据
    while(getline(tf,s))
    {
        //通过 stringstream 的特性处理矩阵中的数据到数组中的转换
        stringstream ss;
        ss << s;
        while(!ss.eof())
        {
            ss >> dtemp;
            singleRow.push_back(dtemp);
        }
        //不知道为什么这样做会多一个数据，所以抛掉最后一个
        singleRow.pop_back();
        d = 0;
        for(i = 0; i < singleRow.size(); i++)
        {
            //计算欧氏距离
            d += pow(singleRow[i] - testData[i], 2);
        }
        for(i;i  < testData.size(); i++)
        {
            //对于没有出现在单词表过的待测试数据，距离为 X-0，所以需要平方
            d += testData[i] * testData[i];
        }
        //计算欧氏距离并存储
        res = (double)sqrt(d);
        EuclideanDistance.insert(pair<int,double>(row, res));
        row++;
        singleRow.clear();
    }
    //对欧氏距离排序，取前 k 个再后续操作
    vector<PAIR> EDistance(EuclideanDistance.begin(),
EuclideanDistance.end());
    sort(EDistance.begin(), EDistance.end(), cmp_by_value);
```

```
    fstream onehot("onehot.txt");
    while(getline(onehot, s))
    {
        for(i = 0; i < s.length(); i++)
        {
            if(s[i] == '1')
            {
                singleRow.push_back(1);
            }
            else if(s[i] == '0')
            {
                singleRow.push_back(0);
            }
        }
        d = 0;
        for(i = 0; i < singleRow.size(); i++)
        {
            //????
            d += pow(singleRow[i] - testData[i], 2);
        }
        for(i;i  < testData.size(); i++)
         {
             d += testData[i] * testData[i];
          }
        res = (double)sqrt(d);
        EuclideanDistance.insert(pair<int,double>(row, res));
        row++;
        singleRow.clear();
    }
```

1） KNN 的关键部分是对于欧氏距离的计算处理，采用欧氏距离的原因是其常用性。由于我采用了 map 来存储欧式距离的结果，而 map 不是线性容器，在 C++中可以把他转化成 vector，再通过 sort 函数编写比较标准来解决这个问题。

当我再每一次处理好欧氏距离的结果后，就对每一个标签求众数，最大者作为预测标签返回。

一开始我是使用 OneHot 矩阵的，由于验收的原因改成了 TF 矩阵。但是最终我还是选择采用 OneHot 矩阵，一方面 OneHot 的效率比较高，一方面我在 OneHot 模型下的准确率明显比较高（大雾）。

```
        double cnt_joy = 0, cnt_sad = 0, cnt_fear = 0, cnt_anger =
0,cnt_surprise = 0, cnt_disgust = 0, cnt_max = 0;
    res_final = 0;
    for(i = 0; i < k; i++)
    {
        cnt_anger +=  (Emotion_anger[EDistance[i].first] /
(EDistance[i].second + 1) );
        cnt_disgust +=  (Emotion_disgust[EDistance[i].first] /
(EDistance[i].second + 1) );
        cnt_fear +=  (Emotion_fear[EDistance[i].first] /
(EDistance[i].second + 1));
        cnt_joy +=  (Emotion_joy[EDistance[i].first] /
(EDistance[i].second + 1));
        cnt_sad +=  (Emotion_sad[EDistance[i].first] /
(EDistance[i].second + 1));
        cnt_surprise +=  (Emotion_surprise[EDistance[i].first] /
(EDistance[i].second + 1));
    }
         }
```

2） 其次是 KNN 的回归算法。回归部分之前的是数据处理和欧氏距离处理的思路同分类算法，不同点在于训练数据的标签变成了情感概率的数据，在这里我的处理是先将同一类情感概率的数据（列）存储在同一个 vector 里，且不打乱训练数据的 id 顺序，这样在最后求出 K 个数据的时候只需要根据【欧氏距离所对应的行号】取寻找训练数据的【id】，在情感概率的 vector 里找出对应的概率，除以欧氏距离即可。为了避免这里出现分母为 0 的情况（因为可能出现待测试数据与训练数据的欧氏距离为 0）.所以在分母统一加上 1 以避免算数异常，不影响最终结果。

3.2 NB 部分

```
typedef struct Train {
    vector <double> words;
    string label;
    vector <double> emotions;
}Train;
```

（vector string）ArticleSet=所有训练文本，不包含情感标签
（vector string）DataSet=单词表，不重复
（vector double）testData=当前待测试文本的数据（待测试数据，只有一行）
（struct）Train=所有训练文本的每一句（包括矩阵处理后的数据、情感标签、情感概率）
（vector double）cnt_true[ ]=一个待测试文本的情感预测数组，用于存储情感预测的概率。

```
One_Hot();
    fstream onehot("onehot.txt");
    //改进训练文本的存储方式
    int row = 0;
    //由于 NB 的多项式模型使用的是频次的计算，所以使用 onehot 矩阵
    while (getline(onehot, s))
    {
        Train singleRow;
        for (i = 0; i < s.length(); i++)
        {
            if (s[i] == '1')
            {
                singleRow.words.push_back(1);
            }
            else if (s[i] == '0')
            {
                singleRow.words.push_back(0);
            }
        }
        singleRow.label = Emotion_train[row];
        eachRow.push_back(singleRow);
        row++;
    }
```

3） 文件的预处理部分如上图所示，相比于 KNN 有所改进。

```
    double cnt_all[6] = { 0 };
    vector <double> cnt_true[6];
    string cnt_label[6] = {"joy","sad","fear","anger","surprise","disgust"};
    for (i = 0; i < testData.size(); i++)
    {
        if (testData[i] == 1)//只根据 1 去寻找有效的对象
        {
            for (i1 = 0; i1 < 6; i1++)//反复 6 次寻找 6 种情感标签的所有频率
            {
                temp = 0;
                for (j = 0; j < eachRow.size(); j++)//所有训练文本
                {
                    if (eachRow[j].label == cnt_label[i1])//当训练文本的情感标签符合第 i1 种
                    {
                        temp += count(eachRow[j].words.begin(), eachRow[j].words.end(),
1);
                            if (eachRow[j].words[i] == 1)//对应位置位 1 则有效
                            {
                                cnt_all[i1]++;
                            }
                    }
                }
                cnt_true[i1].push_back(double( (cnt_all[i1]+1) / (temp +
DataSet.size())));
            }
        }
    }
```

4）上图是 NB 算法的分类部分，思路是根据待测试数据中（已经处理为 OneHot 矩阵之后）的 1 的位置，再去所有的训练数据中寻找对应的位置是否为 1，若为 1 则该位置是一个有效投票，计数；每一次寻找训练数据都要统计该训练数据的有效 1 的个数，以此作为统计的有效投票的除数。该部分结束之后得到了【一个测试数据】的【6 种情感的概率】，这 6 个概率之后再取最大值即可得出对应的最终预测的标签。

```
    vector <double> cnt_true[6];
    string cnt_label[6] = { "anger","disgust","fear","joy","sad","surprise" };
    for (i1 = 0; i1 < 6; i1++)//循环 6 次寻找所有情感标签的概率
    {

        for (j = 0; j < eachRow.size(); j++)//对于每一个文本
        {
            temp = 1;
            for (i = 0; i < eachRow[j].words.size(); i++)
            {
                if (testData[i] != 0)//只根据非 0 去寻找有效的对象
                {
                    //无论对应位置是否为 0，都要相乘
                    temp *= (eachRow[j].words[i] + a) / (eachRow[j].words.size() -
count(eachRow[j].words.begin(), eachRow[j].words.end(),0) + a * DataSet.size())  ;
                }
            }
            temp *= eachRow[j].emotions[i1];//该文本第 i1 种情感的概率
            cnt_true[i1].push_back(temp);
        }
        for (j = 0; j < cnt_true[i1].size(); j++)
        {
            cnt_all[i1] += cnt_true[i1][j];
        }
        cnt_max += cnt_all[i1];
    }
    //对测试文本的六种情感预测结果进行归一化处理
    for (i = 0; i < 6; i++)
    {
        cnt_all[i] = cnt_all[i] / cnt_max;
    }
```

4） 以上代码是 NB 的回归部分，回归相比于分类，在循环的部分有所不同。由于同样是根据测试数据中的非零位置去到训练数据对应的位置进行相乘的操作，所以循环外部套的第一层就是对于【6 个情感标签】，去寻找【每一个训练数据】，在该位置的【第 i1 种情感的概率】，相乘下一个对应位置的【第 i1 种情感的概率】，全部算完之后再乘以该训练数据的【第 i1 种情感标签的数据】，得到该训练数据条件下待测试数据的【i1 种情感标签的概率】，循环算完所有训练数据，就可以得到再所有训练数据条件下待测试数据的【i1 种情感标签的概率】，再进行加和操作即可得到【i1 种情感标签的总概率】，6 次循环就可以得到【所有情感标签的总概率】。归一化处理后返回处理结果即可。

5） 其余的函数我认为不是很重要，没有必要再写出来分析了。一方面也节省 TA 的时间。简洁地概括一下，就是每一次读入测试文本，转换成测试数据（矩阵）后，按照每一行（单个测试文本）的读取去调用 KNN 或 NB 的分类或回归算法，因为返回的是 string 或 double 数组，再把这个结果进行处理（输出或存储）即可。

4. 创新点&优化（如果有）

1） 优化：使用拉普拉斯平滑。。

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）

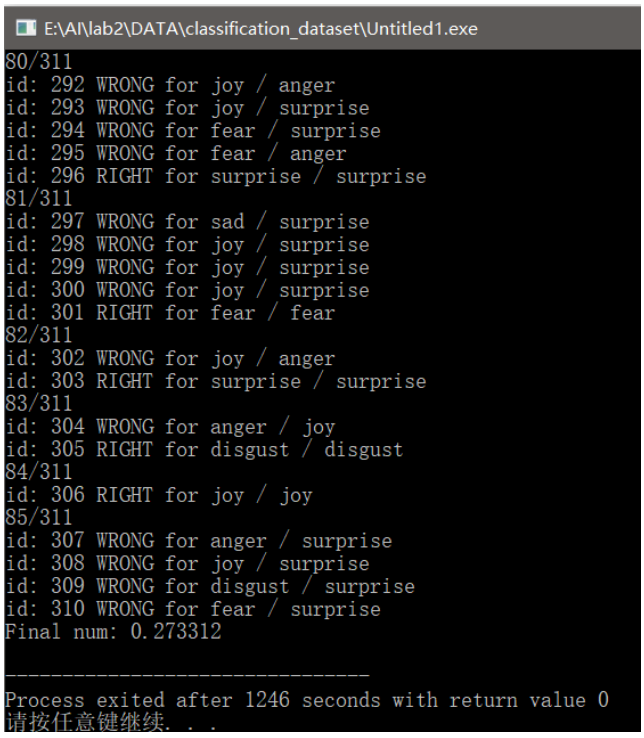


图 1 KNN 分类（TF）验证集（k=14）

1） KNN 采用 TF 跑验证集的结果。

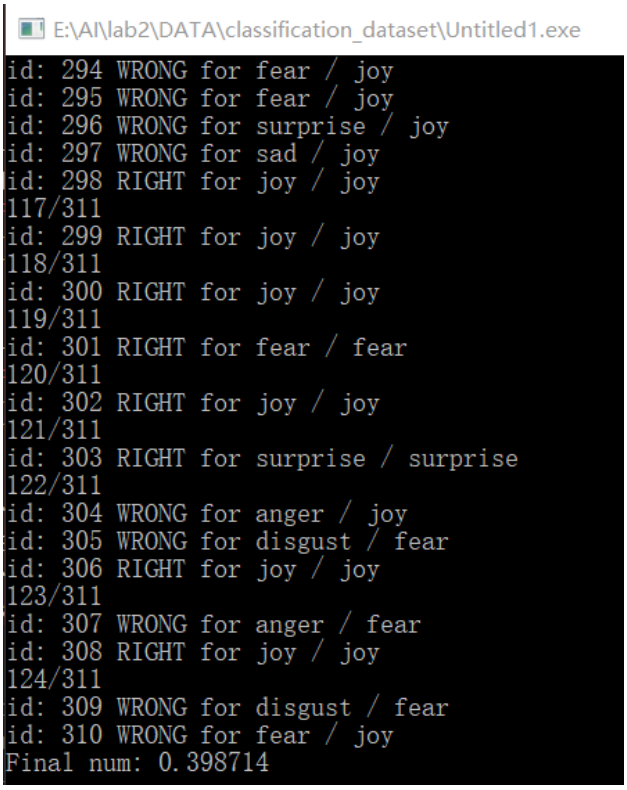
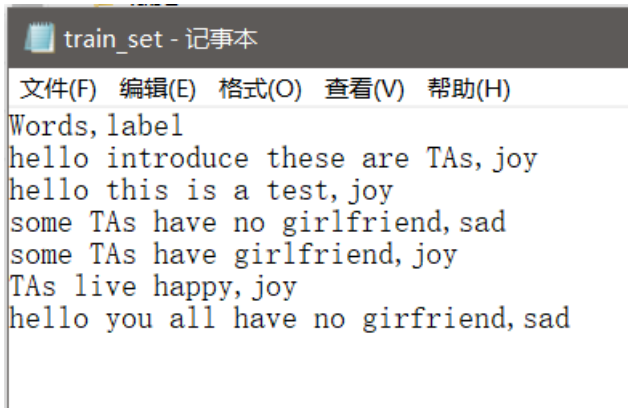
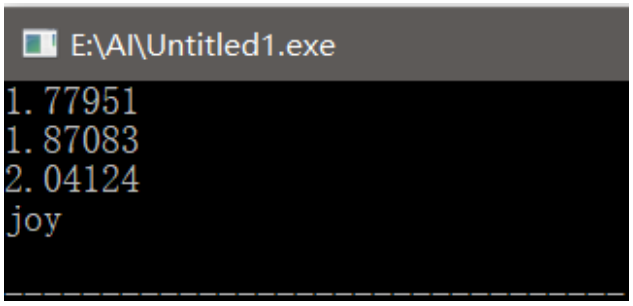


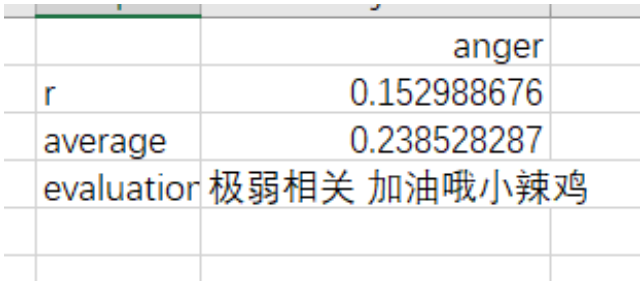
图 2 KNN 分类（OneHot）验证集（k=14）

2） KNN 采用 OneHot 跑验证集分类的结果，速度明显提升。



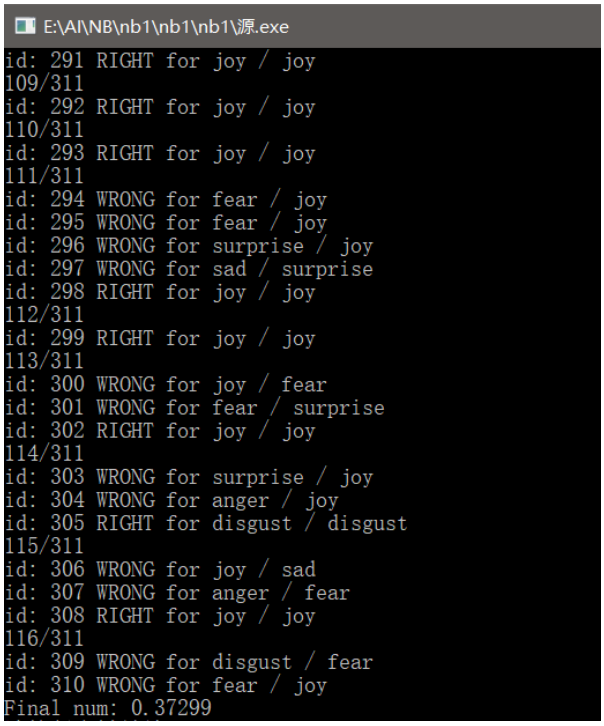


3） 在该训练文本的条件下当 k=3 时得出的结果为 joy，虽然有点悲伤，但是苦中作乐嘛，人丑就应该多读书，也可以的，经过验证，确实应该是这样。



	anger
r	0.152988676
average	0.238528287
evaluation	极弱相关 加油哦小辣鸡

4） KNN 回归的结果，由于时间原因没有深入去考虑为什么回归出来的结果并不是非常理想。



5） NB 分类的验证集结果，拉普拉斯平滑稀疏 a=0.004。稍微比 KNN 低一点。

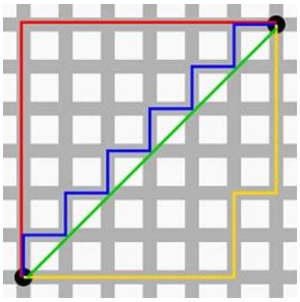
	I	J
		anger
	r	0.320941665
	average	0.371587013
	evaluation	低度相关 666

6） NB 回归的验证集结果，可以看到效果比 KNN 好得多。

四、 思考题

1. 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？

首先，欧氏距离通过计算二者的整体距离（不相似性）得到评判二者的相似程度，但是在实际应用中仍然有比较大的缺点。单纯从公式上考虑，二者的主要区别就是 p=1 和 p=2 的区别。若从计算性能上考虑，曼哈顿距离可能不需要浮点数运算（在使用 OneHot 矩阵的情况下）。当矩阵稀疏程度较大时，曼哈顿距离和欧式距离的表现效果差距较大（个人猜测）。因为参考百度的一张图，如下，红线曼哈顿距离，绿线欧氏距离，可以看出如果这个矩阵正方形的规模较小，那么红线和绿线的差距就很接近，但是规模大的时候二者的相差就比较远了，由此猜测在稀疏矩阵程度较大时，使用曼哈顿距离和使用欧氏距离计算出来的结果的结果误差程度会比较大。



（实在不知道怎么回答了，请 TA 高抬贵手。。。）

2.伯努利、多项式两个模型分别有什么优缺点？

优点：伯努利模型更容易实现，多项式模型由于加入了出现频率作为考量标准，精确度更高。

缺点：伯努利模型的精确度较低（在单词表规模较大的时候）、多项式模型计算量更加复杂。

3. 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？

本次实验中，我的处理方法是：

KNN：将没有出现过的词加入到词典中，并且在计算欧氏距离的时候将没有出现过的词直接当作与训练数据的距离为最大（当前距离-0）处理。也就是说，待测试集中有一个之前没有出现过的单词，那么二者的距离就应该适当加大，而不是忽略。

NB：直接忽略掉没有出现过的词，不进入频率的计算范畴。因为不论加还是不加，训练文本中都不会出现这个词，所以在计算频次的时候不管怎么进行拉普拉斯平滑这个结果依然很小，最终还是会影响到整体的频率使其趋向 0，所以干脆舍去。

|---------- 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