中山大学移动信息工程学院本科生实验报告

（2015 年秋季学期）

课程名称：Artificial Intelligence　　任课教师：饶洋辉

年级	13 级	专业（方向）	移动信息工程
学号	13354485	姓名	朱琳
电话	13726231932	Email	280273861@qq.com

一 说明

本次试验实现了 6 种算法——1NN，NB，ID3,C4.5,PLA,LR。是个整合型的报告。

二 代码实现部分

(一)数据处理部分

【说明】所有的实验代码均用以下数据结构，并且读取文件的函数没有基本没有发生改变。因此不再复述。

1.数据结构

```
char data[] = "Datac_all.csv";
char result[] = "result.txt";
vector<vector<float> > trains, tests;
int row_of_trains = 0, row_of_tests = 0;
float dis[12000][28000];
```

Trains 和 tests 的数据用一个 vector<vector<float> >来表示，其中里面的 vector 用来盛放每一行的数据，外面的 vector 的 size 就是整个训练集或测试集的数量。

2.读取文件

```
void read() {
    ifstream fin(data);
    if (fin == NULL)
        cout << "cannot read file";
    string line;
    getline(fin, line); //读取一行
    while (!fin.eof()) {
        getline(fin, line); //读取一行
        if (line.length() == 0)
            break;
        if (line.substr(line.length() - 1, 1) == "?") {
            stringstream rfloat; //read float
            line = line.substr(0, line.length() - 1);
            rfloat << line;
            vector<float> testVec;
            while (!rfloat.eof()) {
                float n;
                char dot;
                rfloat >> n >> dot;
                testVec.push_back(n);
            }
            tests.push_back(testVec);
            row_of_tests++;
```

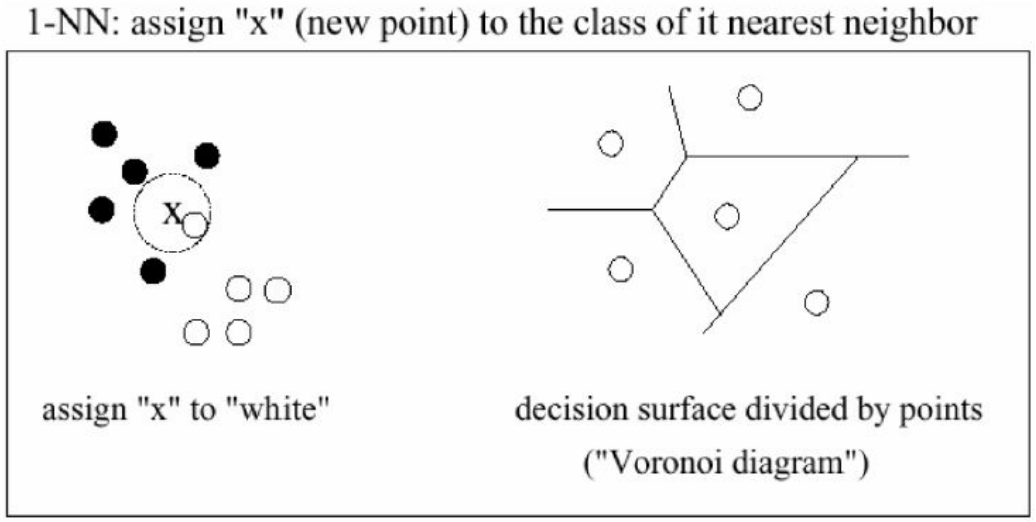
```
        } else {
            line += ",";
            stringstream rfloat; //read float
            rfloat << line;
            vector<float> trainVec;
            while (!rfloat.eof()) {
                float n;
                char dot;
                rfloat >> n >> dot;
                trainVec.push_back(n);
            }
            trains.push_back(trainVec);
            row_of_trains++;
        }
    }

    cout << "row_of_trains=" << row_of_trains << endl;
    cout << "row_of_tests =" << row_of_tests << endl;
}
```

(二)主要算法代码部分

【1.kNN】

【算法分析】kNN，即 k 近邻算法，是将测试文本与训练文本的属性值作比较，选择距离最近的 k 个邻居，然后根据这些邻居最后的结果进行加权来预测测试实例的结果，距离可采用欧氏距离，曼哈顿距离，余弦距离等。关于 kNN,之前做过两次实验，也比较熟悉了,大致流程就是读取文本->获取文本向量->计算各个测试文本对于各个训练文本之间的距离->选取最小（余弦距离选最大）那个距离->利用距离大小进行加权，距离越近，相似度越高，权重越大->通过权重和训练实例的结果值来预测测试实例的结果。



(1)得到文本的距离——经过反复试验，采用曼哈顿距离作为最终的选择。

```
void get_dis() {
    for (int ttRow = 0; ttRow < row_of_tests; ttRow++) { //每个测试文本
        for (int tnRow = 0; tnRow < row_of_trains; tnRow++) { //每个训练文本
            float distance = 0;
            int cols = trains[tnRow].size();
            for (int k = 0; k < cols; k++) {
                distance += abs(trains[tnRow][k] - tests[ttRow][k]);
            }
            dis[ttRow][tnRow] = distance;
        }
    }
}
```

(2)kNN 具体计算部分

```
void kNN() {
    for (int ttRow = 0; ttRow < row_of_tests; ttRow++) { //对于每个测试文本
        float min = INT_MAX;
        for (int tnRow = 0; tnRow < row_of_trains; tnRow++) {
            if (dis[ttRow][tnRow] < min) {
                int cols = trains[tnRow].size();
                int shares = trains[tnRow][cols - 1];
                min = dis[ttRow][tnRow];
                tests[ttRow].push_back(shares);
            }
        }
    }
}
```

【kNN 算法优化部分】因为这次只要求实现 1NN,因此我简化了原来的代码。开始我采用余弦距离并且没有归一化，得到的结果是 0.56，后来对文本属性的每一列进行了[0,1]映射，即用归一化公式 value= (value-min)/(max-min).通过遍历文本，得到某属性的最大值和最小值，再次循环遍历文本，使用上述公式进行归一化。具体代码如下：

```
//进行向量列归一化
void vector_to_one() {
    int cols = trains[0].size() - 1;
    for (int col = 0; col < cols; col++) { //对于每一列来说，除去最后一列
        float max = -999999, min = 999999;
        for (int row = 0; row < row_of_trains; row++) { //for each row
            max = trains[row][col] > max ? trains[row][col] : max;
            min = trains[row][col] < min ? trains[row][col] : min;
        }
        for (int row = 0; row < row_of_tests; row++) { //for each row
            max = tests[row][col] > max ? tests[row][col] : max;
            min = tests[row][col] < min ? tests[row][col] : min;
        }
        for (int row = 0; row < row_of_trains; row++) { //for each row
            trains[row][col] = (trains[row][col] - min) / (max - min);
        }
        for (int row = 0; row < row_of_tests; row++) { //for each row
            tests[row][col] = (tests[row][col] - min) / (max - min);
        }
    }
}
```

归一之后采用曼哈段距离实现效果最好，为 0.58

1	0	0	1	请在左边“B”列中，粘贴你的模
2	0	0	1	你的模型预测准确率如下：
3	0	1	0	0.58
4	0	0	1	
5	0	0	1	

【算法思考】

感觉用交叉验证的方式确定 N 值的效果一定不错，但是这需要花费大量的时间来确定 N 值，这样会大大影响运行速度。

【2.NB】

【算法分析】

naive Bayesian，朴素贝叶斯算法。通俗来说，就是对于给出的待分类项，求解在此项出现的条件下各个类别出现的概率，哪个最大，就认为此待分类项属于哪个类别。因为 lab5 这个 NB 跟 lab2 的 NB 相差甚远，重新写了一遍朴素贝叶斯的实现，同时理了理思路。

naive Bayesian 定义如下：

1、设$x = \{a_1, a_2, ..., a_m\}$为一个待分类项，而每个 a 为 x 的一个特征属性。

2、有类别集合$C = \{y_1, y_2, ..., y_n\}$。

3、计算$P(y_1|x), P(y_2|x), ..., P(y_n|x)$。

4、如果$P(y_k|x) = max\{P(y_1|x), P(y_2|x), ..., P(y_n|x)\}$，则$x \in y_k$。

现在的关键就是如何计算第 3 步中的各个条件概率。我们可以这么做：

①找到一个**已知分类的待分类项集合**，即**训练样本集**。

②统计得到在**各类别下各个特征属性**的条件概率估计。即
$P(a_1|y_1), P(a_2|y_1), ..., P(a_m|y_1); P(a_1|y_2), P(a_2|y_2), ..., P(a_m|y_2); ...; P(a_1|y_n), P(a_2|y_n), ..., P(a_m|y_n)$。

PS：根据 lab5 数据集，只有两种分类，0or1；

③如果各个特征属性是条件独立的，则根据贝叶斯定理有如下推导：

$$P(y_i|x) = \frac{P(x|y_i)P(y_i)}{P(x)}$$

因为分母对于所有类别为常数，因为我们只要将分子最大化即可，又因为各特征属性是条件独立的，所以有：

$$P(x|y_i)P(y_i) = P(a_1|y_i)P(a_2|y_i)...P(a_m|y_i)P(y_i) = P(y_i)\prod_{j=1}^{m} P(a_j|y_i)$$

由于 lab5 的数据集是连续变量，因此我们使用高斯分布来确定$P(a_j|y_i)$

总结上述模型的大致流程如下：

确定特征属性->获取训练样本->对每个类别计算 P(Yi)->对每个特征属性计算所有划分的条件概率->对每个类别计算 P(Xi|Yi)*P(Yi)->用 P(Xi|Yi)*P(Yi)最大项作为 x 所属类别。

【绿色部分】是进行分类器训练，任务就是生成分类器，即计算每个类别在训练样本中的出现频率及每个特征属性划分对每个类别的条件概率估计，并将结果记录。其**输入**是**特征属性**和**训练样本**，输出是分类器。【橘色部分】是应用阶段，任务是使用分类器对待分类项进行分类，其**输入**是**分类器**和**待分类项**，输出是待分类项与类别的映射关系。

【代码优化部分】因为需要使用到高斯分布，因此我们需要计算均值和方差，我采用公式 V=E(x^2）-E(x)^2 来计算，避免了频繁的平方和相加减的操作。同时，因为得到的条件概率可能很小，我采用了取对数的方式来防止数据下溢出。另外，为了防止出现$P(a_j|y_i)$ =0 的情况，采用拉普拉斯平滑来进行校准。

```
float E[100],Epow2[100], V[100];
for (int col = 0; col < cols; col++) { //对于训练文本的每一列
    E[col] = 0, V[col] = 0;
    for (int tnRow = 0; tnRow < row_of_trains; tnRow++) { //对于每个

        E[col] += trains[tnRow][col];
        Epow2[col]+=trains[tnRow][col]*trains[tnRow][col];
    }
    E[col] /= row_of_trains;
    V[col]=abs(E[col]*E[col]-Epow2[col]);//计算方差V=E(x)^2-E(x^2)
    float coeffcient=1.0/(sqrt(2.0*PI*V[col]));
    float b=2.0*V[col];
    float LogPy=0;
    for (int tnRow = 0; tnRow < row_of_trains; tnRow++){
        float t=trains[tnRow][col]-E[col];
        float Py=coeffcient*exp(t*t/b);//获取高斯分布概率
        LogPy+=log(Py);
    }

}
```

【**最终结果**】最终得到的结果为 0.58。

你的模型预测准确率如下：		
0.58		

NB 有一个缺点，NB 模型假设属性之间相互独立，但是这个假设在实际情况中甚至在本次实验中并不总是成立的，这给 NB 模型的正确分类带来了一定影响。

【3&4.决策树—ID3&C4.5】

【算法解析】

D3 算法的核心思想就是以信息增益度量属性选择，选择分裂后信息增益最大的属性进行分裂。

增益是由信息熵进行计算的，具体公式如下：

$$Gain(S,A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

但是 ID3 算法存在一个问题，就是偏向于多值属性，比如按照 key 值进行分类，那么就会出现过拟合的现象。ID3 的后继算法 C4.5 使用增益率（gain ratio）的信息增益扩充，试图克服这个偏倚。增益率用如下公式计算：

$$split\_info_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} log_2(\frac{|D_j|}{|D|}) \qquad gain\_ratio(A) = \frac{gain(A)}{split\_info(A)}$$

决策树首先需要算熵值，算增益（增益率），然后根据增益(率)选取节点，判断是否可以区分开所有训练集，如果可以就停止迭代，如果不行，则需要进行继续扩大树结构。必要时因为属性过多需要进行剪枝操作。

(1)计算熵值和增益，并选择最大增益的节点

```
int ID3(nestVec data, StrIntMap is_used) {
    int row = data.size();
    int select_col = -1;
    int col = data[0].size();
    map<string, double> D;
    //计算d的熵值
    for (int i = 0; i < row; i++) {
        if (D.find(data[i][col - 1]) != D.end()) {
            D[data[i][col - 1]]++; //若之前存在这个属性，直接++;
        } else {
            D[data[i][col - 1]] = 1; //初始化属性值。
        }
    }
    double HD = 0; //H(D)
    map<string, double>::iterator it;
    for (it = D.begin(); it != D.end(); it++) {
        HD += -1 * D[it->first] * 1.0 / row * log2(D[it->first] * 1.0 / row);
    }
    //计算Gain(D;A);
    double GainDA = 0;
    for (int i = 0; i < col - 1; i++) {
        if (is_used[attrName[i]] == 1)
            continue;
        map<string, double> A;
        for (int j = 0; j < row; j++) {
            if (A.find(data[j][i]) != A.end()) {
                A[data[j][i]]++;
            } else {
                A[data[j][i]] = 1;
            }
```

```
        }
        double HDA = 0; //计算条件熵值
        for (map<string, double>::iterator it = A.begin(); it != A.end();
                it++) {
            map<string, double> AD;
            double rowA_sum = 0;
            for (int j = 0; j < row; j++) {
                if (data[j][i] == it->first) {
                    if (AD.find(data[j][col - 1]) != AD.end()) {
                        AD[data[j][col - 1]]++;
                    } else {
                        AD[data[j][col - 1]] = 1;
                    }
                    rowA_sum++;
                }
            }
            for (map<string, double>::iterator it_AD = AD.begin();
                    it_AD != AD.end(); it_AD++) {
                HDA += A[it->first] / row * (-1 * AD[it_AD->first] / rowA_sum)
                        * log2(AD[it_AD->first] / rowA_sum);
            }
        }
        //选择增益最大的那个数作为节点
        if (GainDA < HD - HDA) {
            GainDA = HD - HDA;
            select_col = i;
            cout<<"Gain(D;A)="<<GainDA<<endl;
        }
    }
    return select_col;
}
```

(2)判断是否是叶子节点，即判断此树是否能够进行所有训练集的正确分类

```
string judge_leaf(nestVec data) {
    int row = data.size();
    int col = data[0].size();
    StrIntMap D_num;
    for (int i = 0; i < row; i++) {
        if (D_num.find(data[i][col - 1]) != D_num.end()) {
            D_num[data[i][col - 1]]++;
        } else {
            D_num[data[i][col - 1]] = 1;
        }
    }
    string res;
    int max_num = -1;
    for (StrIntMap::iterator it = D_num.begin(); it != D_num.end(); it++) {
        if (it->second > max_num) {
            res = it->first;
            max_num = it->second;
        }
    }
    return res;
}
```

(3)建树

```
173 void build(Node * &root, nestVec data, StrIntMap is_used) {
174     StrIntMap res;
175     int col = data[0].size();
176     int rows = data.size();
177     for (int i = 0; i < rows; i++) {
178         if (res.find(data[i][col - 1]) != res.end()) {
179             res[data[i][col - 1]]++;
180         } else
181             res[data[i][col - 1]] = 1;
182     }
183     if (res.size() == 1) {
184         root = new Node;
185         root->s = data[0][col - 1];
186         return;
187     }
188     int select_col = ID3(data, is_used);
189     if (select_col == -1) {
190         root = new Node;
191         root->s = judge_leaf(data);
192         return;
193     } else {
194         string selected_attribute = attrName[select_col];
195         is_used[selected_attribute] = 1;
196         root = new Node;
197         root->s = selected_attribute;
198         int child_size = node[selected_attribute].size();
199         nestVec temp_data[child_size];
200         int j = 0;
201         for (Vector::iterator it2 = node[selected_attribute].begin();
202                 it2 != node[selected_attribute].end(); it2++, j++) {
203             int rows = data.size();
204             for (int i = 0; i < rows; i++) {
```

```
204             for (int i = 0; i < rows; i++) {
205                 if (data[i][select_col] == *it2) {
206                     temp_data[j].push_back(data[i]);
207                 }
208             }
209         }
210         int real_child = 0;
211         for (j = 0; j < child_size; j++) {
212             if (temp_data[j].size() != 0) {
213                 real_child++;
214             }
215         }
216         if (real_child > 1) {
217             j = 0;
218             for (Vector::iterator it2 = node[selected_attribute].begin();
219                     it2 != node[selected_attribute].end(); it2++, j++) {
220                 if (temp_data[j].size() == 0) {
221                     root->child[*it2] = new Node;
222                     root->child[*it2]->s = judge_leaf(data);
223                 } else {
224                     root->child[*it2] = NULL;
225                     build(root->child[*it2], temp_data[j], is_used);
226                 }
227             }
228         } else {
229             root->s =judge_leaf(data);
230         }
231         is_used[selected_attribute] = 0;
232     }
233 }
```

(4)C4.5 中增益率的计算—只是在 ID3 的基础上添加几行代码。

```
        //A的熵值
        double HA = 0;
        for (map<string, double>::iterator it = A.begin(); it != A.end();
                it++) {
            HA += -1 * (it->second) / row * log2((it->second) / row);
        }
        double ratio = (HD - HDA) / HA;
        cout<<"GainRatio="<<ratio<<endl;
        //选择ratio最大的那个数作为节点
        if (GainDA < ratio) {
            GainDA = HD - HDA;
            select_col = i;
            //cout<<"GainRatio="<<ratio<<endl;
        }
    }
```

【算法分析】最终得到的结果如下

你的模型预测准确率如下：		
0.61		

【5.PLA+五折交叉验证】

【算法分析】

PLA，感知机学习算法，使用一个 vector W 来对所有的训练实例的 X 进行加权，加权之后得到的数的符号可用来判断结果，如果结果正确，则判断下一个训练实例，如果不正确则需要使用下列公式进行迭代:

$$\mathbf{w}_{(t+1)} \leftarrow \mathbf{w}_{(t)} + y_{n(t)}\mathbf{x}_{n(t)}$$

其中 Yn 有两种取值为 1 和-1。由符号函数来判断。得到更新的 W 之后再次进行迭代，在理想情况下迭代到可以正确区分所有的训练集为止。在大数据集下，我们几乎不可能有一个 w 满足所有的分类，因此上述的方式算法根本无法停止，因此我们需要选择合适的迭代次数，并且最好实现五折交叉验证来选取最好的 W 值。另外 W 的初值由自己决定，一般先确定训练实例的维度，再进行 W 的相应初始化，本次试验我将 W 的值初始化为全 1。

```
void PLA() {
    for (int c = 0; c < times; c++) {
        for (int row = 0; row < Train_row_4_5; row++) { //共有row_of_trains个训练样本
            double y = 0;
            int num_of_cols = Trains[row].size() - 1;
            for (int col = 0; col < num_of_cols; col++) {
                y += Trains[row][col] * w[col]; //计算此刻对于文本的预测值
            }
            int real_sign = sign(Trains[row][Trains[row].size() - 1]);
            if (sign(y) != real_sign) { //若预测失败
                int col = 0;
                int row_of_w = w.size();
                for (int rw = 0; rw < row_of_w; rw++, col++) {
                    w[rw] += real_sign * Trains[row][col];
                }
                //break;
            }
        }
```

【优化部分——交叉验证】以训练集 4/5 作为训练文本，后 1/5 作为测试文本，验证得到效果最好的 W 值。

```
    //交叉验证
    if (c > 100 && c % 5 == 0) {
        int count = 0;
        for (int tr = Train_row_4_5; tr < row_of_tests; tr++) {
            int col2 = Trains[tr].size();
            double res = 0;
            for (int col = 0; col < col2; col++) {
                res += Trains[tr][col] * w[col];
            }
            int result = 1.0 / (1.0 + exp(-res)) >= 0.5 ? 1 : 0;
            int real_result = Trains[tr][col2 - 1];
            if (result == real_result)
                count++;

        }
        //若正确率比之前的都高，那么就将w放到w_store队列里面去。
        if (count > final_count) {
            final_count = count;
            w_store.push(w);
        }
    }
}
```

【解释】我是设定了一个栈 w_store()来专门存取进行交叉验证之后的 w 的值的。每迭代 10 次，就进行一次验证，如果此时验证的正确率大于原来的正确率，就将此事的 w push 到 w_store 中去，在最后对测试文本进行验证的时候，再将 w_store 中顶部的 w 拿出来作为最终的 w 即可。

```
void print() {
    fstream fout(result);
    fout.clear();
    vector<double> ww = w_store.top();
    int Wsize = ww.size();
    for (int row = 0; row < row_of_tests; row++) {
        double t=0;
        for (int col = 0; col < Wsize; col++) {
            t += Tests[row][col] * ww[col];
        }
        cout<<t<<endl;
        int sign_=sign(t)>0?1:0;
        fout << sign_ << endl;
    }
}
```

【最终结果】0.63

	A	B	C	D	E
1	0	0	1	请在左边“B”列中，粘贴	
2	0	0	1	你的模型预测准确率如下：	
3	0	1	0	0.63	
4	0	0	1		
5	0	0	1		

【6.LR】

【算法解析】

Logic Regression，逻辑回归算法，其实就是 $W_{new}=W_{old}-n*\nabla$，其中∇是梯度，n 是步长。这个公式可以演化得到：

$$\tilde{\mathbf{W}}_{new}^{(j)} = \tilde{\mathbf{W}}^{(j)} - \eta \frac{\partial C(\tilde{\mathbf{W}})}{\partial \tilde{\mathbf{W}}^{(j)}}$$

$$= \tilde{\mathbf{W}}^{(j)} - \eta \sum_{i=1}^{n} \left[\left(\frac{e^{\tilde{\mathbf{W}}^{\mathrm{T}} \tilde{\mathbf{X}}_i}}{1+e^{\tilde{\mathbf{W}}^{\mathrm{T}} \tilde{\mathbf{X}}_i}} - y_i \right) \tilde{\mathbf{X}}_i^{(j)} \right]$$

这本次实验中我先用一些常规的做法来实现。有实现进行向量列归一化，LR 实现部分如下：

```
void LR() {
    for (int c = 0; c < times; c++) {
        int wcol = w.size();
        double error[wcol];
        for (int i = 0; i < wcol; i++)
            error[i] = 0;
        for (int row = 0; row <row_of_trains ; row++) { //y=w0+(sigma(1->n))Wj*Xj
            double wx = 0;
            int num_of_cols = Trains[row].size();
            for (int col = 0; col < num_of_cols - 1; col++) { //-1是去掉结果的那一列
                wx += w[col] * Trains[row][col]; //Wj*Xj;
            }
            double diff = 1.0 / (exp(-wx) + 1) - Trains[row][num_of_cols - 1]; //(  -Yi)*Xi
            for (int col = 0; col < num_of_cols - 1; col++) {
                error[col] += diff * Trains[row][col]; //*Xi部分
            }
        }
        for (int wc = 0; wc < wcol; wc++)
            w[wc] -=step * error[wc];
    }
}
```

结果如下：

0	0	1	请在左边“B”列中，粘
0	0	1	你的模型预测准确率如
0	0	1	0.59
0	0	1	
0	0	1	
1	1	1	

【LR 五折交叉验证部分】

```
void cross_validation() {//实现交叉验证
    cout << "validation" << endl;
    int count = 0;
    int wcol=w.size();
    for (int tr = Train_row_4_5; tr < row_of_trains; tr++) {
        int result = predict(Trains[tr],w);
        int real_result = (int)(Trains[tr][wcol]);
        if (result == real_result) {
            count++;
        }
    }
    //若正确率之前的高，那么就将w放到w_store队列里面去。
    cout<<"count="<<count;
    if (count >= final_count) {
        final_count = count;
        cout<<"final="<<final_count<<endl;
        w_store.push(w);
    }
}
```

结果是：

1	0	0	1	请在左边“B”列中，粘贴你的模
2	0	0	1	你的模型预测准确率如下：
3	0	0	1	0.58

我试着调了下训练集合的比例，并没有什么用，反而更小了。然后我将每次更新的结果输出出来，发现 w 迭代到一定的地步就可以预测成功所有的训练集（作为测试集）的文本。看来是过拟合了。

二 实验结果比较

	kNN	NB	决策树	PLA	LR
最好结果	0.58	0.58	0.61	0.63	0.59

综上，PLA 最终的实现效果最好。