一、 实验题目，

1. 数据的读写。
2. 计算 one_hot 矩阵，TF 矩阵和 TF_IDF 矩阵。
3. 计算 one_hot 矩阵的三元组矩阵。
4，稀疏矩阵加法运算。

二、 实验内容

1. 算法原理

文件可以读入到程序中，通过处理，可以得到一个词汇表。每个文件中的词语都可以在词汇表中找到，根据每个文件出现的词汇，我们可以计算 one_hot 矩阵，TF 矩阵和 TF_IDF 矩阵，也可以把矩阵表现出三元组矩阵的形式。得到的矩阵是稀疏矩阵，我们可以实现矩阵的相加，同样，也是用三元组矩阵的形式表现。

2. 伪代码

function vaculary：

从文件中取出一行
把单词分割出来
单词放入一个 vector 中
去除重复单词，得到词汇表

function one_hot：

查看词汇表中的单词是否出现在这一行
是则输出 1，并且记录此单词在这一行出现次数
否则输出 0，并且记录此单词在这一行出现次数为 0

function TF：

得到这一行单词的总个数
每个单词在这一行出现次数除以这一行单词总个数，保留小数

function TF_IDF：

计算有某个单词的文件的个数
计算这个单词的 IDF
乘上这个单词的 TF 值，得到 TF_IDF 矩阵

function smatrix:

计算文件总行数
计算词汇表大小
计算稀疏矩阵中非零值个数
得到它的三元组：
只有值为非零的数需要在这里体现
第一个数为行号

第二个数为在词汇表中的顺序
第三个数为这个词汇出现总次数

function AplusB:

把上面的三元组矩阵平均分成两份
两个稀疏矩阵按照行列相加
最终得到的矩阵表现成三元组矩阵形式

3. 关键代码截图（带注释）

把词语从一行当中取出：

```
//从第二个tab之后就是词汇了
int tab = line.find('\t',10);
//把词汇从一整句中分离
str = line.substr(tab + 1,line.length() - tab - 1)+" ";

while(str.length() != 0)
{
    int blank = str.find(" ");
    file[file_num].push_back(str.substr(0,blank));
    s.push_back(str.substr(0,blank));
    str = str.erase(0,blank + 1);
}
```

```
//得到词汇表c
c.push_back(s[0]);
for(int i = 1; i < s.size(); i++)
{
    flag = 0;
    for(int j = 0; j < i; j++)
    {
        find_sum++;
        if(s[i] == s[j])
        {
            flag = 1;
            break;
        }
    }
    if(flag != 1 && s[i] != "\n")
    {
        c.push_back(s[i]);
    }
}
```

One_hot 矩阵的生成：

```
//假如某个词汇出现，则输出1，否则输出0
//且记录这个词汇在该文件中出现次数，方便TF矩阵计算
for(int i = 1; i <= file_num; i++)
{
    for(int j = 0; j < c.size(); j++)
    {
        if(in_file(c[j],i) != 0)
        {
            one_hot<<"1"<<" ";
            word_num++;
            one_hot_[i].push_back(in_file(c[j],i));
        }
        else
        {
            one_hot<<"0"<<" ";
            one_hot_[i].push_back(0);
        }
    }
    one_hot<<endl;
}
```

TF 矩阵的生成：

```
//输出TF矩阵之后，同时放进TF_这个vector记录，方便下面计算TF_TDF矩阵
for(int i = 1 ; i <= file_num; i++)
{
    double word_sum = file[i].size();

    for(int j = 0; j < c.size(); j++)
    {
        if(word_sum != 0)
        {

            TF_[i].push_back((double)one_hot_[i][j] / word_sum);

            TF<<(double)one_hot_[i][j] / word_sum<<" ";
        }

    }
    TF<<endl;
}
```

计算 TF_IDF 矩阵：

```
//计算词汇出现在多少个文件中，暂时没有计算出IDF
void word_in_file_func()
{
    for(int i = 0; i < c.size(); i++)
    {
        for(int j = 1 ; j <= file_num; j++)
        {
            for(int k = 0; k < file[j].size(); k++)
            {
                if(c[i] == file[j][k])
                {
                    word_exit_in_file[i]++;
                    break;
                }
            }
        }
    }
    return;
 }
```

```
for(int i = 1; i <= file_num; i++)
{
    for(int j = 0; j < c.size(); j++)
    {
        TF_IDF << TF_[i][j] * log((double) file_num / word_exit_in_file[i])<<" ";
     }
     TF_IDF<<endl;
 }
```

输出稀疏矩阵的三元组矩阵：

```
//输出稀疏矩阵三元顺序表
 void smatrix_func()
 {
    ofstream smatrix("smatrix.txt");
    //前三个数为文件数量、词汇表大小和one_hot矩阵中1的总数
    smatrix << file_num << endl;
    smatrix << c.size() << endl;
    smatrix << word_num << endl;

    for(int i = 1; i <= file_num; i++)
    {
        for(int j = 0; j < one_hot_[i].size(); j++)
        {
            if(one_hot_[i][j] != 0)
            {
                smatrix << i - 1 << " " << j << " "<< one_hot_[i][j] << endl;
             }
         }
     }

 }
```

AplusB：

```
int not_zero = 0;
for(int i = 1; i <= file_num / 2; i++)
{
    int j = file_num / 2 + i;
    for(int k = 0; k < c.size(); k++)
    {
        //值为非零则相加
        if(one_hot_[i][k] != 0 || one_hot_[j][k] != 0)
        {
            smat[i-1].push_back(one_hot_[i][k] + one_hot_[j][k]);
            not_zero++;
        }
        //否则，值为0
        else
        {
            smat[i-1].push_back(0);
        }
    }
}
```

4. 创新点&优化（如果有）

因为直接用 vector 来做词汇表，去重时候需要查找很多遍（n 的平方次），所以我决定用哈希表重新做一遍，这样，查找次数应该可以降为 n。如果文件大一点，这应该是很好的优化。

果然，没有用哈希表时候的查找次数为 **15288099**，文件词汇一共有八千多个，这也就是 n 的平方次查找。但是经过哈希表的应用之后，查找次数一下子就降下来了，次数变成了 **8861**，大约就是 n 次查找而已，词汇表创建速度一下子升上去了。

这里用的哈希函数是比较经典的 ELFHash 函数。

```
unsigned int ELFHash(const char *str)
{
    unsigned int hash = 0;
    unsigned int x = 0;

    while (*str)
    {
        hash = (hash << 4) + (*str++);
        if ((x = hash & 0xF0000000L) != 0)
        {
            hash ^= (x >> 24);
            hash &= ~x;
        }
    }
    return (hash & 0x7FFFFFFF);
}
```

它适用于字符串的哈希表建立，能够有效避免冲突。

如果发生冲突，这里采用的是拉链法解决。

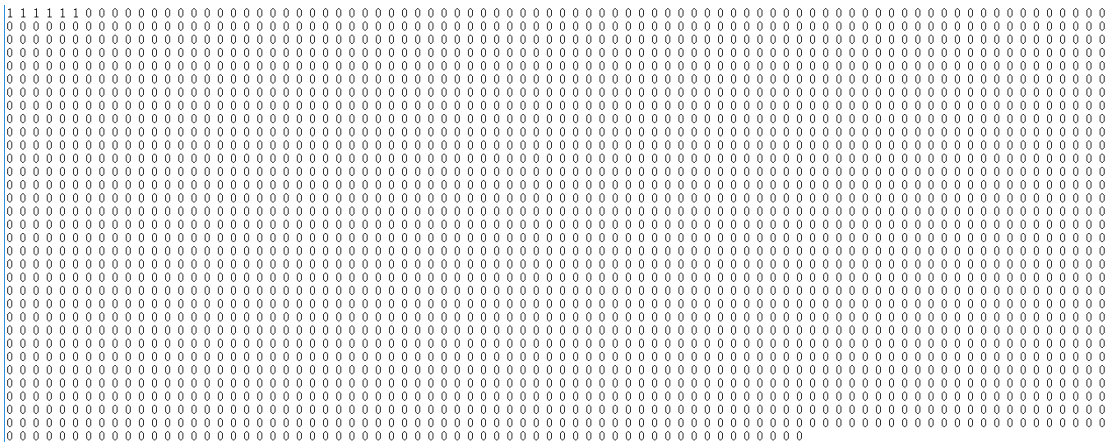
```
tempA = HashTable[key];
//如果这个值被占用，那就用拉链法，也就是在原有值的指针指向新的值
while (tempA != NULL)
{
    if (!strcmp(tempA->str, str))
        return;
    tempB = tempA;
    tempA = tempA->next;
}

tempA = (Node *)malloc(sizeof(Node));

tempA->str = str;
tempA->next = NULL;
tempB->next = tempA;
```

经过哈希表之后，我们就有了查找每个单词的编号（哈希函数产生的 key 值）了，所以字符串查找的时间大大缩减。

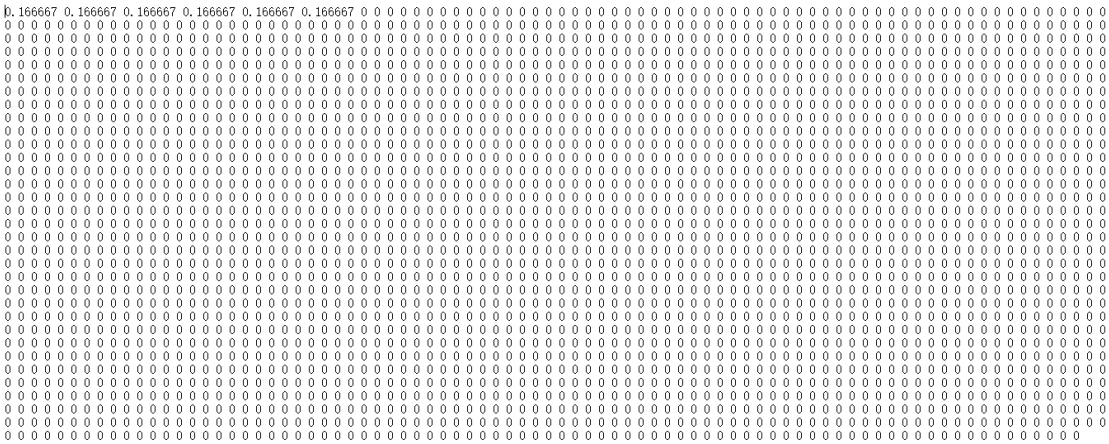
三、 实验结果及分析

1. 实验结果展示示例（只是为了展示矩阵的形式，截图很小，TA 没必要认真看每个数值）

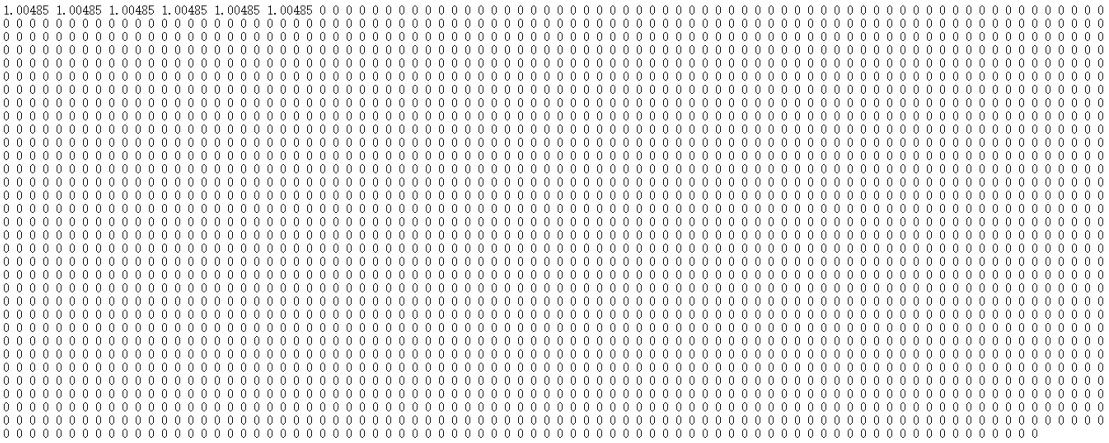
one_hot 矩阵为一个 1246 行，2749 列的矩阵。矩阵是一个稀疏矩阵，也就是数值多为 0.非零的值即是 1.



TF 矩阵也是一样大小的稀疏矩阵，矩阵的非零值是小于等于 1 大于 0 的小数。



TF_IDF 矩阵大小相同，同样为稀疏矩阵。非零值是小数但是可能大于 1.



Smatrix 矩阵前三行只有 1 列，后面紧跟着 8189 行，每行有 3 列。.

```
1246
2749
8189
0 0 1
0 1 1
0 2 1
0 3 1
0 4 1
0 5 1
1 6 1
1 7 1
1 8 1
1 9 1
2 5 1
2 10 1
2 11 1
2 12 1
2 13 1
2 14 1
3 15 1
3 16 1
3 17 1
4 18 1
4 19 1
4 20 1
4 21 1
4 22 1
4 23 1
5 24 1
5 25 1
5 26 1
5 27 1
5 28 1
5 29 1
6 13 1
6 30 1
6 31 1
6 32 1
```

AplusB 矩阵的前三行也是只有 1 列，后面有 8098 行，每行三列，.

```
623
2749
8098
0 0 1
0 1 1
0 2 1
0 3 1
0 4 1
0 5 1
0 45 1
0 300 1
0 514 1
0 515 1
0 1736 1
0 1781 1
0 1782 1
1 6 1
1 7 1
1 8 1
1 9 1
1 416 1
1 775 1
1 1197 1
1 1783 1
1 1784 1
2 5 1
2 8 1
2 10 1
2 11 1
2 12 1
2 13 1
2 14 1
2 131 1
2 311 1
2 902 1
2 1785 1
2 1786 1
2 1787 1
3 10 1
3 15 1
3 16 1
3 17 1
```

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）

(原始算法结果)

|----------如有优化，请重复 1，2，分析优化后的算法结果----------|

算法优化主要是加快词汇表生成。

原先的查找次数：

```
Search 15288099 times.
```

使用哈希表之后的查找次数：

```
Search 8861 times.
```