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）	**软件工程（移动信息工程）**
学号	**15352033**	姓名	**陈黄胤**

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文件读入

使用文件流 fstream 读取文件 semeval，然后将流中的内容利用 getline 整行存储至字符串 std::string 中,然后再利用 substr 读取每行两个’/t’之后的句子内容，再利用 stringstream 字符串流对句子进行分词处理，存储于单词 vector 之中，单词 vector 又被存储于句子 vector 之中，对于单词以及句子的其他属性存储，则利用结构体扩展其存储信息。

②one-hot 矩阵求解

读取每个句子的单词的时候就通过一个单词总表，用于判断获取每个单词的出现顺序（即 one-hot 矩阵的索引位置），然后在单词结构体中加入这样的一个 index 变量。所以求解 one-hot 的时候只需要遍历句子数组，然后遍历每个句子中的单词数组中每个单词的 index，将其存在桶 bucket（大小为单词总表的大小）对应的下标位置，若句子存在着这个单词 index 就将 bucket[index]标为 1，最后遍历一遍 bucket 即可得到这一行的 one-hot 矩阵值，继续遍历下一个句子，直到每一条句子的 one-hot 值都被输出到文件中。

③TF 矩阵求解

优化一下单词结构体的存储信息，以及单词数组的存储方式。改为计数制，即这个句子中已经出现过这个单词就将单词的计数器 num++，若没出现过则新建一个 word 结构体并将 num 置为 1 然后再插入到单词数组。此时根据先前的方式来遍历每一个句子的每一个单词数组。令 bucket[index]=word[index].num，最后遍历 bucket 输出即可得到这一行的 TF 矩阵值，继续遍历下一个句子，直到每一条句子的 TF 矩阵值都被输出到文件中。

④TFIDT 矩阵求解

优化一下读入时的操作，新建一个 WORD_NUM 的 vector 用于存储单词出现在了多少个句子中。若是在整一个文件中第一次出现这个单词，则新建一个计数器(int)归零后 push_back 进去 WORD_NUM，若是第一次出现在这个句子中，则对应下标的计数器++，这样就得到了出现了该单词的文章总数。通过公式，将第三步中的 TF 矩阵乘以 log(B/(1+WORD_NUM))，然后逐行输出即可。

⑤SMATRIX 矩阵求解

首先，先输出句子的条数（行数），单词表的总大小（列数），以及每一个句子中不重复的单词个数和。

再利用 one-hot 中的求解过程，把 one-hot 中的遍历过程中的所有句子中的每一个不重复单词的行、列输出, 三元组的值在这里显然都为 1。

⑥AplusB 三元组矩阵加法的实现

首先先读入两矩阵 A 和 B 的行列以及三元组个数。显然，只有行列值都相等的三元组矩阵才能相加并且得到一个行列值仍然一致的新矩阵，这里不考虑错误输入的情况，所以直接输出了任意一个矩阵的行、列值。加法的实现我们首先定义两个索引值 aindex 和 bindex，用于指示 A、B 两个矩阵当前的加法运算情况。由于我们输入的两个三元组矩阵都是按升序排好序的，于是直接对所有可能的行列值按照从 0 开始的升序进行搜索。对每一个可能的行列值组合(依照输入的矩阵行列范围)，对 A、B 两矩阵的对应 aindex/bindex 处进行访寻，如果匹配到了 A 中的，再访寻 B 中的当前 index 是否匹配，若两者都匹配，则将两者的 value 值相加后，建成新的三元组组合插入到结果矩阵 C 中，若只有其中一个匹配，则直接复制这一个匹配的三元组插入到矩阵 C 中，若没有匹配，则继续循环。直到所有可能都遍历完毕，将矩阵 C 输出到文件中。

2. 伪代码

```
Function read_data()
{
    Define 文件流  in
    in 载入文件 seveval
    while(in)
    {
        将 in 中的字符串输出保存为 string
        string 载入字符串流，作分词处理，用 vector<word>存储起来
        存储入 vector<word>的同时，用 vector<sentence>存储句子，句子中的 word 要存好在总表中的索引 index，以及在这个句子中的出现次数 num,以及 sentence 本身要存好这句中的单词总数 length。
    }
```

```
}

Function get_one_hot()
{
    For  循环遍历 vector<sentence>
    {
        Define int bucket[3000];
        将 bucket 全部置零
        For  循环遍历这个 sentence 的 word_list，将 bucket 中的对应下标位置标 1
        {
            bucket[sentence[i].word_list[j].index]=1;
        }
        将 bucket 中的值输出即可
    }
}

Function get_tf()
{
    For  循环遍历 vector<sentence>
    {
        Define int bucket[3000];
        将 bucket 全部置零
        For  循环遍历这个 sentence 的 word_list
        {
            bucket[sentence[i].word_list[j].index]=
            sentence[i].word_list[j].num/sentence[i].length;
        } //将这个词出现的次数除以句子的总词数即可
        将 bucket 中的值输出即可
    }
}

Function get_tfidf ()
{
    For  循环遍历 vector<sentence>
    {
        Define int bucket[3000];
        将 bucket 全部置零
        For  循环遍历这个 sentence 的 word_list
```

```
            {
                bucket[sentence[i].word_list[j].index]=
                sentence[i].word_list[j].num/sentence[i].length;
            } //将这个词出现的次数除以句子的总词数即可
            For 循环遍历单词总表
            {
                将 bucket[j]中乘上 log(句子条数 SENTENCE_LIST.size()/(1+wordnum[j]))
                的值输出即可,其中 wordnum[j]是词 j 出现在了多少个句子中
            }
        }
    }

    Function get_smatrix()
    {
        输出句子总数(行数)
        输出单词总表数(列数)
        输出各个句子中不重复单词的个数合(条数)
        For  循环遍历 vector<sentence>
        {
            Define int bucket[3000];
            将 bucket 全部置零
            For  循环遍历这个 sentence 的 word_list，将 bucket 中的对应下标位置标 1
            {
                bucket[sentence[i].word_list[j].index]=1;
            }
            For  循环遍历单词总表 WORD
            {
                如果 bucket[j]==1
                就输出此时的 sentence 编号 i，以及单词编号 j，value 输出为 1
            }
        }
    }

    Function AplusB()
    {
        利用文件流读入两个三元矩阵 A、B 的行 row、列 col、有效数量 num1、num2
        建立一个 struct Triple_point 用于存储三元组的行、列、值
        读入数据得到两个矩阵 vector<Triple_point> A,B;
```

Define int aindex=0,bindex=0;

For(i=0;i<row;i++)

For(j=0;j<col;j++)

{

判断 i、j 是否与 A[aindex]或者 B[bindex]的行列值匹配：

Case1： 既与 A 匹配也与 B 匹配，则新建一个 Triple_point 插入到结果矩阵 vector<Triple_point> ans 中，行列值为 i、j，num 则为 A.num+B.num。最后 aindex++、bindex++。

Case2：只与 A 匹配，不与 B 匹配，则新建一个 Triple_point 插入到结果矩阵 vector<Triple_point> ans 中，行列值为 i、j，num 则为 A.num。最后 aindex++。

Case3：只与 B 匹配，不与 A 匹配，则新建一个 Triple_point 插入到结果矩阵 vector<Triple_point> ans 中，行列值为 i、j，num 则为 B.num。最后 bindex++。

}

}

3. 关键代码截图（带注释）

```
4   struct word{
5       string content;
6       int num; //单词在该句中的出现次数
7       int index; //单词总表中的位置
8       word():content(""),num(0){}
9       word(string _A,int i):content(_A),num(1),index(i){}
10  };
11
12  struct senten{
13      vector<word> word_list;
14      int length;  //有多少个单词
15      senten():length(0){}
16  };
17
18  struct Triple_point{
19      int row,col,num;  //三元组行、列、值
20      Triple_point(int r,int c,int n):row(r),col(c),num(n){}
21  };
22
23  vector<string> WORD; //总单词表
24  vector<int> WORD_NUM; //总表对应下标的单词出现在了多少句中
25  vector<senten> SENTENCE_LIST; //句子总表
```

```
28  void READ_DATA()  //文件读取
29  {
30      fstream in;
31      string GET;
32      in.open("semeval",ios::in);
33
34      int i,j,k;
35      while(in){
36          getline(in,GET);
37          in.get();
38          GET=GET.substr(GET.find("\t")+1);
39          GET=GET.substr(GET.find("\t")+1);  //除去TAB,提取所需句子
40
41          string WORD_DATA;
42          senten SENTEN_DATA;
43          int index;
44          stringstream ss;  //用字符串流分割单词
45          ss << GET;
46          int COUNT=0;  //句子单词计数器
```

```
80  void GET_ONE_HOT()
81  {
82      fstream out;
83      out.open("onehot.txt",ios::out);
84      for(int i=0;i<SENTENCE_LIST.size();i++)
85      {
86          int bucket[3000];               //桶排列，将要输出1的位置标出
87          memset(bucket,0,sizeof(bucket));
88          for(int j=0;j<SENTENCE_LIST[i].word_list.size();j++)   //遍历求解
89          {
90              bucket[SENTENCE_LIST[i].word_list[j].index]=1;
91          }
92          for(int j=0;j<WORD.size();j++)
93          {
94              if(j!=0)
95              out << " ";
96
97              out << bucket[j];
98          }
99          out << endl;
100     }
101     out.close();
102 }
```

```
{
    int bucket[3000];                //桶存储，存储对应总表下标的单词在这个句子中的出现次数
    memset(bucket,0,sizeof(bucket));
    for(int j=0;j<SENTENCE_LIST[i].word_list.size();j++)    //遍历求解
    {
        bucket[SENTENCE_LIST[i].word_list[j].index]=SENTENCE_LIST[i].word_list[j].num;
    }
    for(int j=0;j<WORD.size();j++)
    {
        if(j!=0)
        out << " ";

        out << (double)bucket[j]/(double)(SENTENCE_LIST[i].length);  //输出TF
    }
    out << endl;
```

```
for(int i=0;i<SENTENCE_LIST.size();i++)
{
    int bucket[3000];                //桶存储，存储对应总表下标的单词在这个句子中的出现次数
    memset(bucket,0,sizeof(bucket));
    for(int j=0;j<SENTENCE_LIST[i].word_list.size();j++)    //遍历求解
    {
        bucket[SENTENCE_LIST[i].word_list[j].index]=SENTENCE_LIST[i].word_list[j].num;
    }
    for(int j=0;j<WORD.size();j++)
    {
        if(j!=0)
        out << " ";

        out << (double)(bucket[j]/(SENTENCE_LIST[i].length))*log((double)SENTENCE_LIST.size()/(1+WORD_NUM[j])); //按公式求解tfidf
    }
    out << endl;
```

```
152  void GET_SMATRIX()
153  {
154      fstream out;
155      out.open("smatrix.txt",ios::out);
156      out << "[" <<  SENTENCE_LIST.size() << "]" << endl;
157      out << "[" << WORD.size() << "]" << endl;
158      int smatrix_length=0;
159      for(int i=0;i<SENTENCE_LIST.size();i++)
160      {
161          smatrix_length+=SENTENCE_LIST[i].word_list.size(); //将每个句子出现不同的单词数相加
162      }
163      out << "[" << smatrix_length << "]" << endl;
164      for(int i=0;i<SENTENCE_LIST.size();i++)
165      {
166          int bucket[3000];                //桶排列，将要输出1的位置标出
167          memset(bucket,0,sizeof(bucket));
168          for(int j=0;j<SENTENCE_LIST[i].word_list.size();j++)    //遍历求解
169          {
170              bucket[SENTENCE_LIST[i].word_list[j].index]=1;
171          }
172          for(int j=0;j<WORD.size();j++)
173          {
174              if(bucket[j]==1)
175              {
176                  out << "[" << i << "," << j << ",1" << "]" << endl;
177              }
178          }
```

```
208     for(i=0;i<row1;i++)    //遍历所有可能的行列搭配
209         for(j=0;j<col1;j++)
210         {
211             if(i==A[aindex].row && j==A[aindex].col)
212             {
213                 if(i==B[bindex].row && j==B[bindex].col)  //与矩阵A当前下标匹配也与矩阵B当前下标匹配
214                 {
215                     ans.push_back(Triple_point(i,j,A[aindex].num+B[bindex].num));
216                     //A、B对应值相加插入到结果矩阵ans中
217                     aindex++,bindex++; //两者下标都++
218                 }
219                 else
220                 {
221                     ans.push_back(Triple_point(i,j,A[aindex].num));
222                     //只有矩阵A当前下标匹配，将A此下标下的对应值插入到结果矩阵ans中
223                     aindex++; //A下标++
224                 }
225             }
226             else
227             {
228                 if(i==B[bindex].row && j==B[bindex].col)
229                 {
230                     ans.push_back(Triple_point(i,j,B[bindex].num));
231                     //只有矩阵B当前下标匹配，将B此下标下的对应值插入到结果矩阵ans中
232                     bindex++; //B下标++
233                 }
234             }
235         }
```

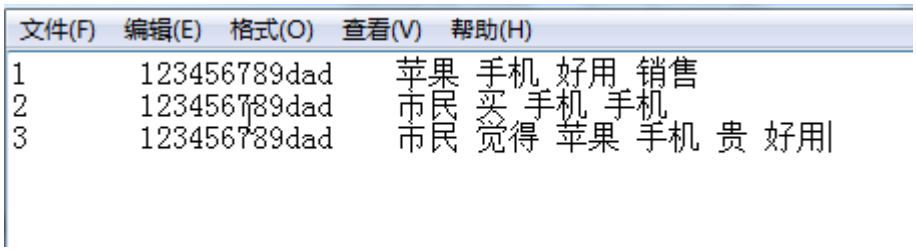
4. 创新点&优化（如果有）

在三元组矩阵加法的实现中，考虑到结果的输出需要按照行号优先列号为次的顺序输出，且由于输入的两个矩阵都是按这个顺序排列好的，可以利用一种类似归并排序的方法。按照输出的需求，循环对比所有的行列组合(先行后列)比较着 A、B 当前的三元组是否与当前循环的行列组合匹配，然后再分情况处理得到最终结果。这种做法大大降低了时间复杂度，不需要像传统做法中一样，每一次探寻都需要先遍历一遍已有的三元组来判断是否已经存在这样的行列搭配了。具体实现可见伪代码和算法原理。

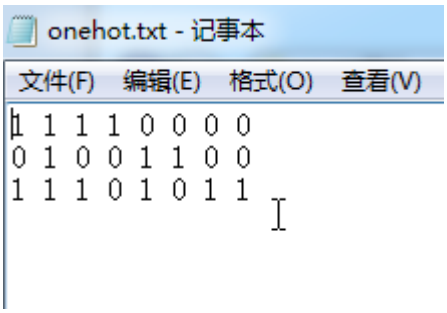
三、 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

测试 1：

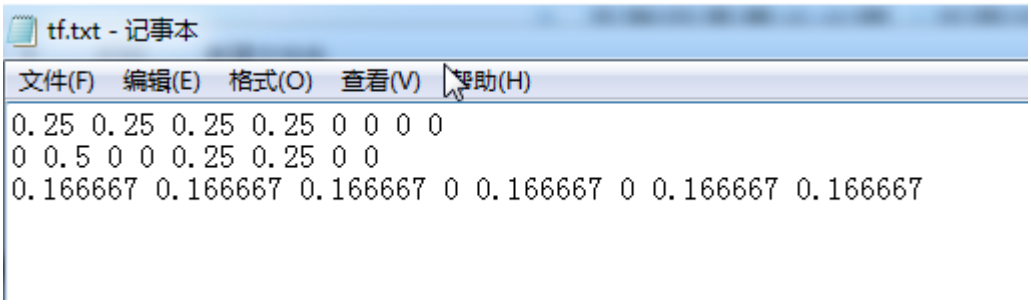


采用实验 ppt 中的测试数据作为小测试集，测试 one-hot、TF、TFIDF、Smatrix 是否正确。其中 one-hot：



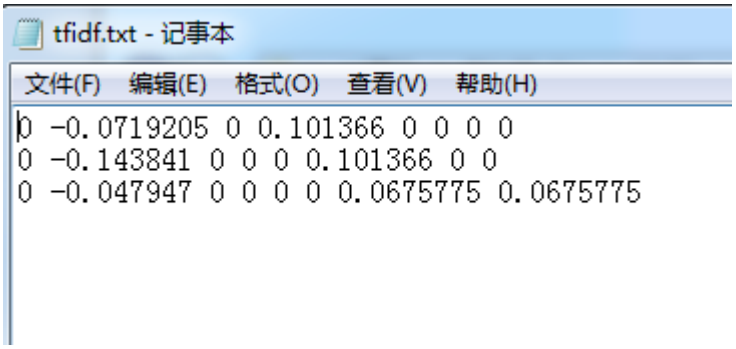
显然正确

其中 TF：



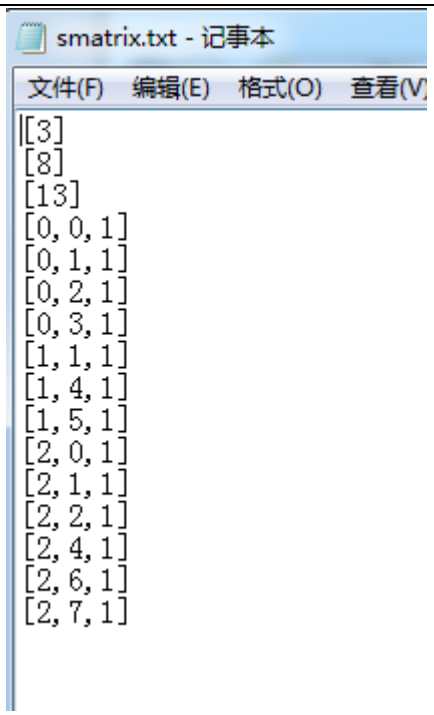
显然也正确。

其中 TFIDF：



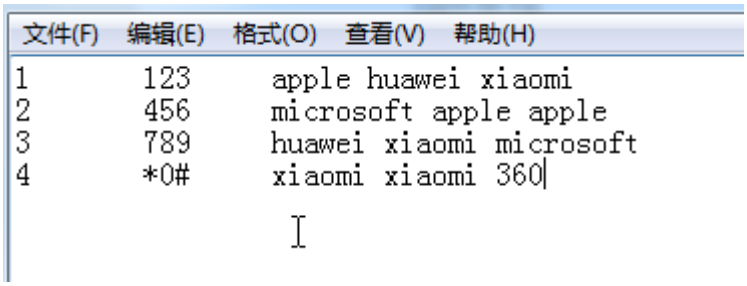
显然也正确

其中 smatrix:



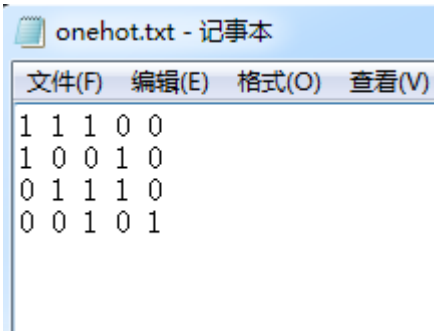
也正确

测试 2：



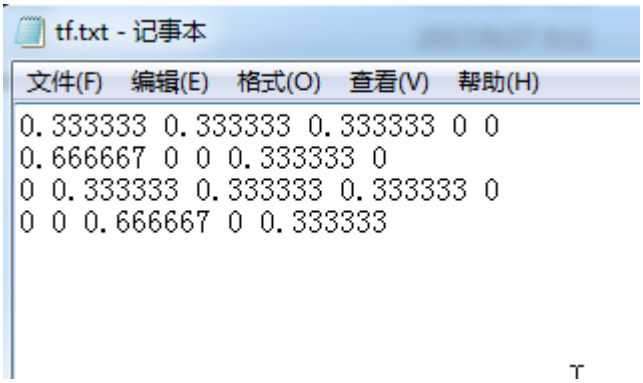
采用上图所示的测试数据作为小测试集，测试 one-hot、TF、TFIDF、Smatrix 是否正确。

其中 one-hot：



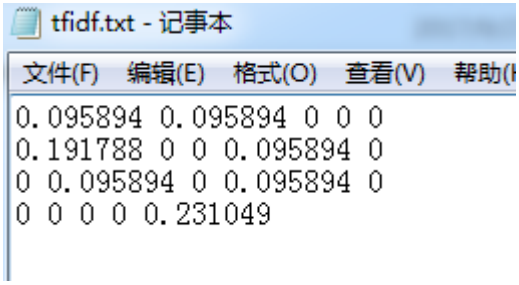
对比发现，结果正确

其中 TF 为：



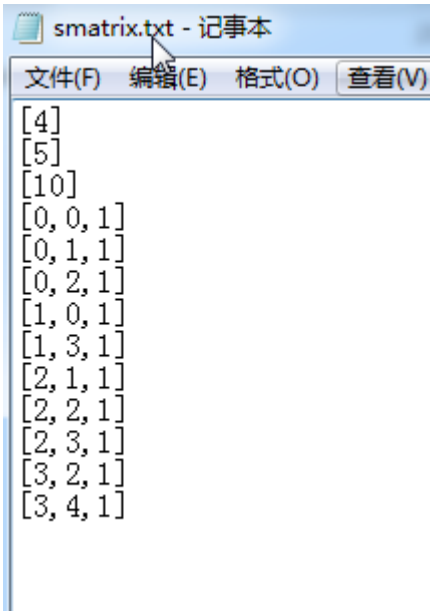
对比发现，结果正确

其中 TFIDF 为：



与手算结果对比，结果正确

其中 smatrix 为：

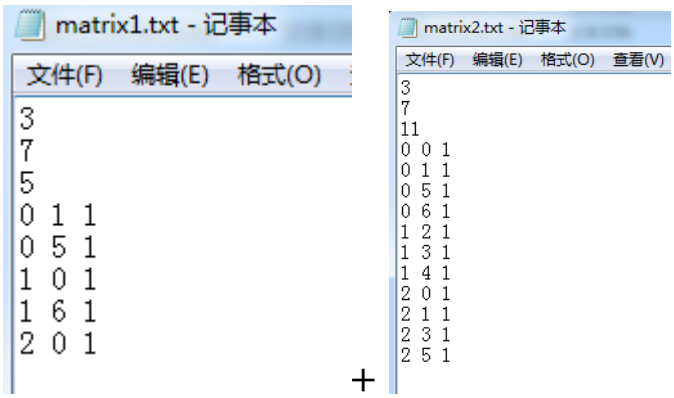


结果正确

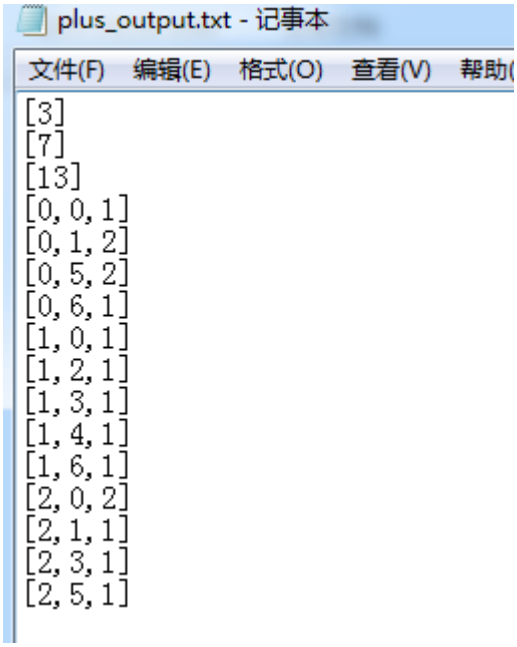
测试 3：

测试 AplusB 函数，看能否实现三元组矩阵加法。

采用以下数据作为测试数据：



最终结果为



发现能够按照行优先列号为次的顺序排列，并且没有遗漏任何一组三元组，结果正确！

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）

One-hot、TF、TFIDF、Smatrix 以及 AplusB 都成功实现

四、 思考题

1. IDF 的第二个计算公式中分母多了个 1 是为什么？

因为出现该单词的文章总数可能为 0，这样的话表达式就不成立了，因为分母不能为 0，所以在计算公式中多了一个 1 可以避免这种情况。

2. IDF 数值有什么含义？TF-IDF 数值有什么含义？

IDF：根据 IDF 公式可知，IDF 可以反映一个单词出现在句子中的频率，越多的句子含有这个词 IDF 越小，而越少句子含有这个词则 IDF 越大，可以理解成一个单词在大量不同文档间的罕见程度，越常见则 IDF 越小，越罕见则 IDF 越大。

TF-IDF：TF-IDF 等于 TF 矩阵乘上 IDF，其中 TF 表示的是在同一个句子中，同一个单词占总单词数的所占比。所以 TF-IDF 表示，一个单词在同一个句子中的频率越高且在所有句子中越罕见则 TF-IDF 值越高。反之，若一个单词在一个句子中的频率低下或者在所有句子中出现频率较高，那么它的 TF-IDF 值会较低

3. 为什么要用三元顺序表表达稀疏矩阵？

因为在一些运算中，存在着高阶矩阵大量元素为 0 的情况，如果用传统的数组来存储就会占用大量的存储空间，也不利于提高某些运算的运算效率(比如矩阵加法，如果用传统的数组方式运算，就需要遍历两个矩阵所有元素相加，其中包含了大量 0 与 0 相加的无用运算；而采用三元顺序表则可以根据算法优化计算，只需要判断对应矩阵位置的情况做出相应处理，只对非 0 元素进行判断运算，就能得到相加后的新三元顺序表，大大提高了效率)。采用三元顺序表，直接存储有效非零数据的行、列、值，有利于提高存储空间利用率，也避免了运算中的很多无用操作，提高了运算效率