

一种自适应信息集成方法

程国达¹, 邹亚会², 朱 静¹

(1. 南京财经大学 信息工程学院, 江苏 南京 210003; 2. 南京财经大学 图书馆, 江苏 南京 210003)

(chengguoda@mail.china.com)

摘 要:检测相似重复记录是信息集成中的关键任务之一, 尽管已经提出了各种检测相似重复记录的方法, 但字符串匹配算法是这些检测方法中的核心。在提出的自适应信息集成算法中, 用一个综合了编辑距离和标记距离的混合相似度去度量字符串之间的相似度。为了避免由于表达方式的差异而造成的字符串之间的不匹配, 字符串被分割成独立的单词后按单词的第一个字符进行排序。在单词的匹配中, 对拼写错误和缩写有一定的容错功能。实验结果表明, 自适应信息集成方法比用 Smith-Waterman 和 Jaro 距离有更高的正确率。

关键词:相似重复记录; 混合相似度; 自适应信息集成; 字符串匹配

中图分类号: TP391.1 **文献标识码:** A

A self-adaptive approach for information integration

CHENG Guo-da¹, ZOU Ya-hui², ZHU Jing³

(1. College of Information Engineering, Nanjing University of Finance & Economics, Nanjing Jiangsu 210003, China;

2. Library, Nanjing University of Finance & Economics, Nanjing Jiangsu 210003, China)

Abstract: Detecting records that are approximate duplicates, but not exact duplicates, is one of the key tasks in information integration. Although various algorithms have been presented for detecting duplicated records, strings matching is essential to those algorithms. In self-adaptive information integration algorithm presented by this paper, the hybrid similarity, a comprehensive edit distance and token metric, was used to measure the similar degree between strings. In order to avoid mismatching because of different expressions, the strings in records were partitioned into vocabularies, then were sorted according to their first character. In the process of vocabularies matching, misspellings and abbreviations can be tolerated. The experimental results demonstrate that the self-adaptive approach for information integration achieves higher accuracy than that using Smith-Waterman edit distance and Jaro distance.

Key words: approximately duplicate records; hybrid similarity; self-adaptive information integration; strings matching

0 引言

在数字图书馆的工作中, 英文信息的自动采集、整理和集成是一项重要工作。由于所采集的信息常常来自文件、数据库和 Web 文件等多个数据源, 这些数据源不仅在模式上存在差异, 而且数据本身也存在如拼写错误、缩写、不同的表达方式等问题, 这些极易导致重复记录的产生。所谓“重复记录”是指那些描述相同客观对象的记录, 它们的存在不仅耗费存储空间, 而且读者在信息检索中需要耗费精力去人工鉴别检索结果。因此, 消除重复记录是数字图书馆自动信息检索系统中的一项重要工作。

在重复记录的检测方法中^[1-3], 字符串匹配方法是其中的核心。目前, 字符串匹配的方法主要有: 基于距离的方法, 例如 N-Gram^[4], Smith-Waterman^[5] 和 Jaro^[6]; 基于标记的相似度的方法, 如 Jaccard 和 TF-IDF^[7]; 基于统计的方法, 如隐马尔科夫模型 HMM^[8]。

当对字符串 s 和 t 进行比较时, 它们之间的 Smith-Waterman 编辑距离是指将 s 转换成 t 时需要进行的插入、删除和替换的次数。编辑距离越小, 表明它们之间相同的部分越

多, 当编辑距离小于给定的阈值时可以认为它们相同或相似。Jaro 是 Smith-Waterman 编辑距离的改进, 其表达式如下:

$$Jaro(s, t) = \frac{1}{3} \left(\frac{|s'|}{|s|} + \frac{|t'|}{|t|} + \frac{|s'| - T_{s', t'} / 2}{|s'|} \right) \quad (1)$$

其中, s' 是 s 中与 t 中的一个子字符串匹配的字符串, t' 是 t 中与 s 中的一个子字符串匹配的字符串, $T_{s', t'}$ 是将 s' 转换成 t' 所需要操作的次数。

N-Gram 是字符串的矢量表示, 矢量的各个部分由字符位置决定, 一般有 bigram (两个字符)、trigram (三个字符) 和 quadgram (四个字符) 的值。

基于标记的相似度距离, 如 Jaccard 相似度 ($\frac{s \cap t}{s \cup t}$), 可以有效地对“Alvaro Mongo”与“Mongo Alvaro”这样的字符串进行匹配, 而 TF-IDF (Term Frequency-Inverse Document Frequency) 则通过统计单词出现的次数和单词的分布去计算由单词组成的字符串之间的相似度。基于统计的方法, 如 HMM 通过训练样本集构造一些专门的 HMM 对姓名或地址等字符串进行匹配。

英语记录中出现的问题主要有:

1) 单词缩写

收稿日期: 2004-08-27; 修订日期: 2004-10-19

作者简介: 程国达 (1962-), 男, 江西景德镇人, 博士, 副教授, 主要研究方向: 数据仓库、数据挖掘、数据清洗; 邹亚会 (1965-), 女, 江苏南京人, 助理馆员, 主要研究方向: 图书情报检索技术; 朱静 (1978-), 男, 江苏南京人, 助理工程师, 主要研究方向: 数据仓库、数据挖掘。

a) 将单词缩写成前缀。例如,将“Proceedings”缩写成“Proc”;

b) 将若干个单词缩写成每个单词的首字母。例如,将“University of California, San Diego”缩写成“UCSD”;

c) 将单词缩写成前缀和后缀的组合。例如,将“Department”缩写成“Dept”。

2) 字母错误

a) 颠倒。例如,将“machine”写成“machien”。

b) 错用。例如,将“machine”写成“machaeen”。

c) 遗漏。例如,将“machine”写成“machne”。

d) 添加。例如,将“machine”写成“machinee”。

3) 单词顺序的多样性

例如,“Mr. Peter Miller”可能会写成“Mr. Miller, Peter”。

在前面介绍的方法中,基于距离的方法只是对字符串中的字符进行一对一的比较,而没有考虑单词的缩写和排列顺序。基于标记相似度的方法需要先对字符串中每个单词进行匹配,Jaccard 相似度虽然与单词的排列顺序无关,但是依赖于对每个单词的匹配。TF-IDF 可以解决由于错误而导致的字符串的不匹配问题,但是它需要对每个单词进行匹配,然后通过统计匹配单词的数量及其分布来判断字符串相似的程度,这种方法有很大的时间复杂度,不适合处理大型数据库和数据仓库。HMM 是一种概率统计方法,由于有较大的时间复杂度,并不适合处理大型数据库和数据仓库。此外,它还是域依赖的(domain-dependent),即模型只针对某个特定的数据对象有效,当对象发生变化时,需要重新训练样本和设计新的模型。为此,本文提出了一种自适应信息集成方法,能够处理前面所述的各种问题,具有很高的匹配准确率。

1 自适应信息集成方法

在自适应信息集成方法中,字符串之间相似度的度量是一个关键。设字符串 $s = s_1 s_2 s_3 \dots s_m$ 和 $t = t_1 t_2 t_3 \dots t_n$, 则 s 和 t 之间的相似度 Sim 用下式计算:

$$Sim = \frac{\sum_{i=1}^k \gamma_i + g_s + g_t}{(|n_s| - n_{s, abbrev} + g_s + |n_t| - n_{t, abbrev} + g_t) / 2} \times \frac{c_{s,t}}{(|l_s| - |n_{s, abbrev}| + |l_t| - |n_{t, abbrev}|) / 2} \quad (2)$$

如果 Sim 大于一个给定的值,就认为 s 和 t 是相似的,当它等于 1 时, s 和 t 完全相等。(2) 式中的 Sim 是一个混合相似度,其中第 1 个因子反映了字符串分割成单词之后的匹配结果,它是一种基于标记的方法。第 2 个因子反映的是在单词匹配过程中得到的相同字符数占总字符数的比例,是基于距离的方法。第 1 个因子反映了字符串中各局部的相似度,而第 2 个因子则反映了字符串整体的相似程度。例如,在两组字符串“Microsoft Ltd”与“Microsoft corp”,“Microsoft Ltd”与“Oracle Ltd”的比较中,如果只用第 1 个因子,则它们具有相同的相似度。如果同时考虑相同字符占总数的比例,则明显第 1 组字符串比第 2 组字符串具有更大的相似度。

在(2)式的第 1 个因子中, $|n_s|$, $|n_t|$ 分别是字符串 s 和 t 包含的英语单词数; $n_{s, abbrev}$ 是 s 中所包含的一组单词数,它们的首字母组合的单词等于 t 中的由大写字母组成的缩写单词; g_s 是 s 中一组单词匹配 t 中全部由大写字母组成缩写字母的次数。 $n_{t, abbrev}$, g_t 与 $n_{s, abbrev}$, g_s 类似。第 1 个因子中的分母是 s 和

t 中应该匹配的单词数的平均值,其中减去了首字母组成一个字符串缩写单词的词组,这样的词组折算成一个单词,因此, s 和 t 的实际长度分别等于 $|n_s| - n_{s, abbrev} + g_s$ 和 $|n_t| - n_{t, abbrev} + g_t$ 。第 1 个因子的分子反映的单词之间的匹配情况,它包括非缩写单词之间的匹配和缩写单词成功匹配次数,用 γ_i 表示非缩写单词匹配时系数大于或等于 0.8 的系数, g_s 和 g_t 是缩写单词成功匹配的次数,考虑到 s 和 t 中均可能存在多个英语单词去匹配另一个字符串中的缩写单词,因此需要同时加上 g_s 和 g_t 。在(2)式的第 2 个因子中, $c_{s,t}$ 是 s , t 相同字符数, l_s , l_t 分别是 s 和 t 的长度,分母是减掉缩写词组 $n_{s, abbrev}$, $n_{t, abbrev}$ 之后 s 与 t 的平均长度。

单词的匹配系数 γ_i 等于:

$$\gamma_i = \frac{l_c}{\min(l_{s,i}, l_{t,j})} \quad (3)$$

其中, $l_{s,i}$, $l_{t,j}$ 分别是 s 和 t 中匹配的第 i 个和第 j 个单词的长度, l_c 是 $l_{s,i}$, $l_{t,j}$ 中所包含公共字符的长度。当 γ_i 大于等于 0.8 时,就认为被匹配的单词是相似的;当 γ_i 等于 1 时,它们精确相等。在单词的输入过程中,单词越长,出错的可能性就越大,反之越小。此外,出错单词中一个字符错误的可能性最大。因此,将相同或相似的匹配系数阈值设置为 0.8 正是考虑到输入单词时出现的各种错误。当然,当单词的匹配系数大于 0.8 但小于 1 时,只是说明它们是相似的,并不意味着它们是相同的,这还取决于其他单词和整个字符串的匹配结果。

字符串 s 和 t 自适应匹配方法包括以下几个部分:

1) 匹配前文所述的第 b) 种缩写,下面的算法是用 s 去匹配 t 。在 s 中查找所有均由大写字母组成的单词,然后在 t 中寻找与其对应的单词组。由于单词“of”存在于一组可能组成缩写单词的词组中,加之其并无实际意义,因此在缩写单词的匹配前删除它。如果缩写单词匹配成功,则将 gs 加 1,再将 s 中匹配成功的缩写单词和 t 中对应的词组删除,使它们不再参加后面的匹配。由于在 t 中同样存在由单词首字母组成的缩写形式,因此,利用该算法去用 t 匹配 s ,计算缩写单词匹配成功的系数 gt 。

```
gs = 0; //缩写匹配成功次数,初始时为 0
delete( t, "of"); //删除 t 中的单词"of"
for( i = 1; i < l_t; i++) {
    //在 s 中查找全部由大写字母组成的单词, l_s 是 s 的长度
    if( ( s[i-1] >='A' && s[i-1] <='Z' ) && ( s[i] >='A' && s[i] <='Z' ) //连续 2 个字符是大写字母
    { for( j = 2; j < l_i; j++) if( s[j] != ',' && s[j+1] != ',' && s[j] != ' ' ) j++; //计算缩写单词的长度
    new( abbrev[j]); //生成长度为 j 的数组
    abbrev <- s[i-1] s[i] ... s[i+j]; //将 s 中从第 i 个字符到第 i+j 个字符存放到 abbrev 中
    words_t = total_voc(t); //统计 t 中单词数
    new( exact[ words_s ]); //生成一个用于存放 t 中每个英语单词首字母的数组
    exact[ words_t ] <- first character of each word in t;
    //将 t 中每个单词的首字母存入数组 exact 中
    r_s = match( abbrev, exact );
    // exact 中是否存在字符串 abbrev
    //如果是, r_s 等于 1, 否则, 等于 0
    if( r_s == 1 ) {
        delete( t, exact ); //删除 t 已经和 s 匹配的单词
        words_t = words_t - j; //t 中剩余的单词数
```

```

gs = gs + 1;
delete( s, abbrev );          //删除 s 中已匹配的缩写单词
i = i + j;                    //将 s 中的指针向前移动 j 个字符
} }

```

2) 将 s 和 t 中所包含的单词按字典顺序排序, 这样可以消除由于顺序不同造成的不匹配问题。下面是先统计 s 中的单词数的算法 $total$, 然后对单词进行排序。依同样的算法统计 t 中的单词数, 并对单词排序。

```

new( begin );                //生成数组 begin 存放 s 中每个单词开始位置
new( end );                  //生成数组 end 存放 s 中每个单词结束位置
num = 0;                    // num 存放已经找到单词数
mark = 0;
for( i = 0; i < ls; i ++ ) {
    if( ( s[i] != ' ' && s[i] != ',' && s[i] != '.' ) &&
        ( mark == 0 ) ) {
        begin[ ++ num ] = i;          //第 num 个单词开始位置
        mark = 1;                    //置标志为 1
    }
    else if( s[i] == ' ' || s[i] == ',' || s[i] == '.' ) &&
        ( mark == 1 ) {
        end[ num ] = i - 1;          //第 num 个单词结束位置
        mark = 0;
    }
}
copy( st, s );              //将 s 复制到 st 中
for( i = 0; i < num - 1; i ++ )          //对单词排序
    for( j = i + 1; j < num; j ++ ) {
        for( k = 0; k < min( st[ end[i] ] - st[ begin[i] ],
            st[ end[j] ] - st[ begin[j] ]; k ++ ) {
            if( st[ begin[i] + k ] < st[ begin[j] + k ] ) {
                exchange( begin[i], begin[j] );
                //交换 begin[i] 与 begin[j] 的值
                exchange( end[i], end[j] );
                //交换 end[i] 与 end[j] 的值
            }
        }
    }
for( i = 0; i < num; i ++ )
    //将 st 中字符串按 begin[i] 和 end[i] 指定的位置复制到 s 中
    copy( s, st( begin[i], end[i] ) );

```

3) 计算 s 和 t 的相似度 Sim 。在 Sim 的计算过程中, 先按 (3) 式匹配 s 和 t 中的每个单词, 然后累加匹配系数大于 0.8 的结果, 再用 (2) 式计算 s 和 t 之间的相似度。例如, 如果匹配 s 中第 i 个单词, 则首先与 t 中的第 i 个单词匹配; 如果匹配的结果小于 0.8, 则与 t 中其他位置的单词进行匹配; 如果与某个单词的匹配系数等于 1 则停止匹配, 否则与 t 中所有的单词匹配, 选出匹配系数最大者, 如果它大于等于 0.8, 则累加到匹配系数中。这样做是因为字符串中的单词排序时如果单词的第 1 个字母发生错误, 则该单词将会被排列到其他位置上。两个单词匹配时, 如果长度不相等, 则用较短的单词去匹配较长的单词, 这是为了匹配上面所述的缩写 a), c) 两种情况。此外, 为了匹配输入时字母颠倒的单词, 在匹配时, 如果对应的字母不同, 则匹配其附近的单词, 但与 Jaro 方法不同。Jaro 方法是当 s 中的字母 s_i 与 t 中的 t_j 匹配时, j 的范围是 $i - H \leq j \leq i + H$ ($H = \min(|s|, |t|)/2$), 它会使 “will” 和 “wild” 会出现矛盾的结果, 即用 “will” 去匹配 “wild” 时, “will” 中的第 4 个字母 “l” 匹配 “wild” 中的 “d” 出现不匹配时会与它的第 3 个字母 “l” 匹配, 最后匹配结果等于 1。而相反, 如果用 “wild” 去匹配 “will” 则字母 “d” 在 “will” 中找不到相同的字母, 最后的结果是 0.75, 造成这种结果的原因是 “wild” 中的字母 “l” 被匹

配了 2 次。为了避免这种情况的发生, 在这里匹配单词时的范围由 Jaro 中的 $i - H \leq j \leq i + H$ 改为如下形式:

$$\max(\alpha + 1, i - H) \leq j \leq \min(l_i, i + H) \quad (4)$$

α 为匹配单词中最后一个成功比较字母的位置, l_i 是 t 的长度。 j 的左边界取 $\alpha + 1$ 与 $i - H$ 中的最大值是为了避免 t 中的字符被比较 2 次的情况, j 的右边界取 l_i 与 $i + H$ 中的最小值是为了避免取到不属于 t 中的字符。为了减少不必要的重复匹配, 匹配字符串中已经成功匹配的单词要删除, 由于单词删除后要移动其他单词, 为了减少操作时间, 只是将匹配后的单词的第 1 个字母置为空格作为删除标记。具体的算法如下:

```

r = 0;                      //匹配系统初始化
cs,t = 0;                  // cs,t 用于保存 s 与 t 中相同字符数
for( i = 0; i <= words_s; i ++ ) {
    if( t[i][0] != ' ' )    //如果 t 中的第 i 个单词没有被删除
        result = compare( ws,i, wt,i );    //匹配 s 和 t 中的第 i 个单词
    if( result < 0.8 ) {
        max = 0; spot = 0;
        for( j = 0; j < words_t; j ++ ) {
            if( t[j][0] != ' ' )    //查找没有被删除的单词
                result = compare( ws,i, wt,j );
            if( result == 1 ) { t[j][0] = ' '; break; }
            //成功匹配的单词打上删除标志
        }
        else if( max < result ) { max = result; spot = j; }
        //保存除 1 之外的最大匹配结果和对应单词
    }
    if( j == words_t && max >= 0.8 )
        { t[j][0] = ' '; result = max; }
    //删除已成功匹配的单词, 保存结果
}
if( result >= 0.8 ) r = r + result;    //累加单词匹配系数
}
Sim = ( r + gs + gt ) / ( words_s + words_t ) * cs,t / ( ls + lt );
//计算 s 与 t 的相似度

```

单词匹配算法 compare 如下:

```

if( ls,i > lt,j ) exchange( si, tj );
// ls,i, lt,j 分别是 s 中第 i 个单词, t 中第 j 个单词的长度
//总是以长度较短的单词去匹配较长的单词
comm = 0;
//保存 s 中的第 i 个单词与 t 中第 j 个单词公共字符串长度
pt = 1;    // t 中第 j 个单词中最后成功比较的字符位置
for( m = 0; m < ls,i; m ++ ) {
    if( s[i][m] == t[j][m] )
        { comm = comm + 1; pt = pt + 1; }
    else if( m == ls,i-1 && s[i][m] == t[j][lt,j] )
        { comm = comm + 1; pt = pt + 1; }
    //如果 s 中的第 i 个单词是之后一个字母,
    //则与 t 中第 j 个单词的最后一个字符比较
    else {
        for( n = max( pointer + 1, i - H ); n <= min( lt, i + H );
            n ++ )
            {
                //匹配邻近字符
                if( s[i][m] == t[j][n] ) {
                    comm = comm + 1;
                    pt = pt + 1;
                    break;
                }
            }
    }
}
cs,t = cs,t + comm;    //累加 s 和 t 中相同字符数
return r = comm / ( ls,i + lt,j );    //计算匹配系数

```

3 算法复杂度及实验结果

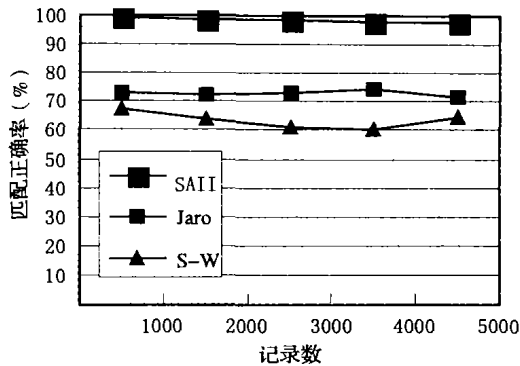


图1 三种匹配方法结果正确率比较

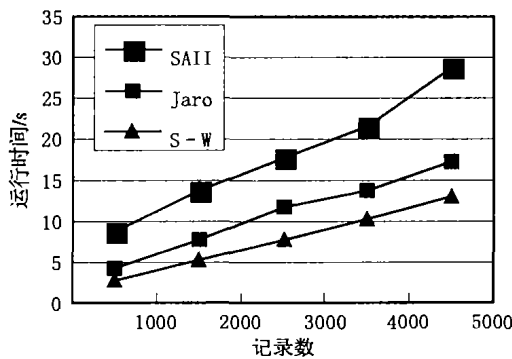


图2 三种匹配方法运行时间的比较

如果字符串 s, t 的长度分别是 $|s|$ 和 $|t|$, 用 Smith-Waterman 编辑距离匹配方法, 则需要操作 (包括比较、删除、插入、复制、替换) 字符的次数等于 $\alpha * \max(|s|, |t|)$ ($1 \leq \alpha \leq 2$), 即当 s 与 t 长度相等并且对应的每个字符相同时, $\alpha = 1$; 如果 s 与 t 没有一个字符相同, 则 $\alpha = 2$ 。因此, 如果以字符操作需要的时间为基本时间单位, 则 Smith-Waterman 编辑距离的时间复杂度等于 $O(\max(|s|, |t|))$ 。

Jaro 所需要的时间 $\beta * (|s| + |t|) + \max(|s|, |t|)$ ($\beta > 1$), 因此, 它的时间复杂度近似等于 $O(|s| + |t|)$ 。假设 s 和 t 包含的单词个数分别为 n_s 和 n_t , 单词的平均长度分别为 a_s 和 a_t , 自适应字符串匹配方法需要的时间是 $2 * (n_s + n_t) + n_s * (n_s - 1) / 2 + n_t * (n_t - 1) / 2 + \min(|s|, |t|)$, 其中, $2 * (n_s + n_t)$ 是分别在 s 和 t 中搜索由首字母组成的单词所花的时间, $n_s * (n_s - 1) / 2, n_t * (n_t - 1) / 2$ 分别是 s 和 t 中单词排序所需要的时间, $\min(|s|, |t|)$ 则是单词匹配所需时间。这是一种理想的情况, 即 s 和 t 中没有首字母缩写组成的单词, 字符串中的单词已经是字典顺序排列而无需重新排列, 此外, 对应的单词也完全匹配。

另一种极端情况是在 s 和 t 中都存在由首字母缩写组成的单词, 而且单词在字符串中的排列也是无序的, 则在 s, t 中搜索由首字母组成单词所需要的时间为 $2 * (n_s - 1) + a_s * (|t| - a_s + 1) + 2 * (n_t - 1) + a_t * (|s| - a_t + 1)$, 单词排序所需要的时间为 $n_s * (n_s - 1) * a_s / 2 + n_t * (n_t - 1) * a_t / 2$, 单词匹配所需要的时间 $\gamma * \min(|s| - \Psi, |t| - \Omega)$, 其中, Ψ 是一组单词, 它的首字母组成 t 中的一个单词。 Ω 也是一组单词, 它的首字母组成 s 中的一个单词。 γ 是一个大于 1 的数, 这是因为对应的单词如果不匹配时, 要搜索 t 中其他位置上的单词, 因此, 当字符串中的单词首字母出现错误时, 实际参加匹配的字符数要大于参加匹配的字符数。

从上面的分析可以看出, 自适应匹配方法的时间复杂度与所匹配的字符串有很大的关系, 它大于 Smith-Waterman 和 Jaro 的时间复杂度, 但小于 $O(\min(|s|^2, |t|^2))$, 这主要是排序和搜索缩写单词所致, 虽然排序与缩写单词的搜索使得计算复杂度增加了, 但却使得算法能有效地匹配各种值, 得到更高的精确度。

为了验证本文自适应信息集成方法 SAI (Self-Adaptive Information Integration) 的有效性, 从 www.computer.org 中的 *magazine*, *transaction* 和 *proceedings* 中收集了论文和论文引用的参考文献中的作者姓名、单位、地址等字符串进行验证, 其中约有 6% 是形式不同的重复数据, 此外还加入了约 10% 的重复数据, 图 1 和图 2 是 SAI 与 Smith-Waterman, Jaro 方法检测数据的结果和运行时间, 从图中可以看出, 文本所提的综合匹配算法要明显优于其他两种匹配方法, 但是时间却比它们要长。以时间的延长换取匹配正确率的提高对那些对运行时间要求无特殊要求的系统来说是非常有意义的。此外, 随着计算机性能的不断提, 处理相同量的数据所需要的时间也会不断地缩短。

4 结语

随着信息技术的发展, 各种信息也爆炸式增长, 人们在信息处理和信息应用的过程中越来越多的遇到信息集成的问题, 而集成后的数据总难以避免存在各种问题, 因此, 研究消除各种问题的数据清洗是非常有意义的。本文提出的自适应信息集成方法有很高的正确率, 因此它对数据清洗中的重复记录消除无疑是很有价值的。

参考文献:

- [1] HERANDEZ MA, STOLFO SJ. The merge/purge problem for large databases [A]. Proceedings of the ACM SIGMOD International Conference on Management of Data [C], 1995. 127 - 138.
- [2] MONGE AE, ELKAN CP. An efficient domain-independent algorithm for detecting approximately duplicate database records [A]. Proceedings of the ACM SIGMOD Workshop on Research Issues on Knowledge Discovery and Data Mining [C]. Tucson, AZ, 1997. 23 - 29.
- [3] BILENKO M, MOONEY RJ. Adaptive duplicate detection using learnable string similarity measures [A]. Proceedings of the 9th ACM SIGMOD International Conference on Knowledge Discovery and Data Mining [C]. Washington, DC, 2003. 39 - 48.
- [4] HYLTON JA. Identifying and merging related bibliographic records [D]. MIT Institute of Technology, 1996.
- [5] SMITH TF, WATERMAN MS. Identification of common molecular subsequences [J]. Journal of Molecular Biology, 1981, 147: 195 - 197.
- [6] JARO MA. Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida [J]. Journal of American Statistical Association, 1989, 86(406): 414 - 420.
- [7] BILENKO M, MOONEY R, COHEN W, et al. Adaptive name matching in information integration [J]. IEEE Intelligent Systems, 2003, 18(5): 16 - 23.
- [8] GENG JF, YANG J. AutoBib: automatic extraction and integration of bibliographic information on the Web [A]. Proceedings of the 29th VLDB Conference [C]. Berlin, Germany, 2003.