相似度

事物的某一性质类似程度。

如果这两句话的用词越相似，它们的内容就应该越相似。因此，可以从词频入手，计算它们的相似程度。

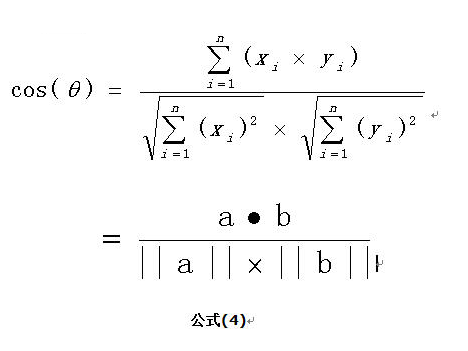
我们就得到了文本相似度计算的处理流程是:

（1）找出两篇文章的关键词；

　（2）每篇文章各取出若干个关键词，合并成一个集合，计算每篇文章对于这个集合中的词的词频

　（3）生成两篇文章各自的词频向量；

　（4）计算两个向量的余弦相似度，值越大就表示越相似。



两个等长字符串之间的汉明距离（英语：Hamming distance）是两个字符串对应位置的不同字符的个数。换句话说，它就是将一个字符串变换成另外一个字符串所需要替换的字符个数。

莱文斯坦距离，又称Levenshtein距离，是编辑距离（edit distance）的一种。指两个字串之间，由一个转成另一个所需的最少编辑操作次数。许可的编辑操作包括将一个字符替换成另一个字符，插入一个字符，删除一个字符。

整數 Levenshtein距離(字符 str1[1..lenStr1], 字符 str2[1..lenStr2])

宣告 int d[0..lenStr1, 0..lenStr2]

宣告 int i, j, cost

對於 i 等於 由 0 至 lenStr1

d[i, 0] := i

對於 j 等於 由 0 至 lenStr2

d[0, j] := j

對於 i 等於 由 1 至 lenStr1

對於 j 等於 由 1 至 lenStr2

若 str1[i] = str2[j] 則 cost := 0

否則 cost := 1

d[i, j] := 最小值(

d[i-1, j ] + 1, // 刪除

d[i , j-1] + 1, // 插入

d[i-1, j-1] + cost // 替換

)

返回 d[lenStr1, lenStr2]

Jaro-Winkler Distance 算法

这是一种计算两个字符串之间相似度的方法，想必都听过Edit Distance,Jaro-inkler Distance 是Jaro Distance的一个扩展，而Jaro Distance（Jaro 1989;1995）据说是用来判定健康记录上两个名字是否相同，也有说是是用于人口普查，具体干什么就不管了，让我们先来看一下Jaro Distance的定义。

两个给定字符串S1和S2的Jaro Distance为：

m是匹配的字符数；

t是换位的数目。

两个分别来自S1和S2的字符如果相距不超过 时，我们就认为这两个字符串是匹配的；而这些相互匹配的字符则决定了换位的数目t，简单来说就是不同顺序的匹配字符的数目的一半即为换位的数目t，举例来说，MARTHA与MARHTA的字符都是匹配的，但是这些匹配的字符中，T和H要换位才能把MARTHA变为MARHTA,那么T和H就是不同的顺序的匹配字符，t=2/2=1.

那么这两个字符串的Jaro Distance即为:

而Jaro-Winkler则给予了起始部分就相同的字符串更高的分数，他定义了一个前缀p，给予两个字符串，如果前缀部分有长度为 的部分相同，则Jaro-Winkler Distance为：

dj是两个字符串的Jaro Distance

是前缀的相同的长度，但是规定最大为4

p则是调整分数的常数，规定不能超过0.25，不然可能出现dw大于1的情况，Winkler将这个常数定义为0.1

这样，上面提及的MARTHA和MARHTA的Jaro-Winkler Distance为：

dw = 0.944 + (3 \* 0.1(1 − 0.944)) = 0.961

以上资料来源于维基百科：

http://en.wikipedia.org/wiki/Jaro-Winkler\_distance

lucene中实现代码分析：

Java代码 收藏代码

public class JaroWinklerDistance implements StringDistance {

private float threshold = 0.7f;

private int[] matches(String s1, String s2) {

String max, min;

if (s1.length() > s2.length()) {

max = s1;

min = s2;

} else {

max = s2;

min = s1;

}

// 两个分别来自s1和s2的字符如果相距不超过 floor(max(|s1|,|s2|) / 2) -1, 我们就认为这两个字符串是匹配的, 因此，查找时，

// 超过此距离则停止

int range = Math.max(max.length() / 2 - 1, 0);

// 短的字符串, 与长字符串匹配的索引位

int[] matchIndexes = new int[min.length()];

Arrays.fill(matchIndexes, -1);

// 长字符串匹配的标记

boolean[] matchFlags = new boolean[max.length()];

// 匹配的数目

int matches = 0;

// 外层循环，字符串最短的开始

for (int mi = 0; mi < min.length(); mi++) {

char c1 = min.charAt(mi);

// 可能匹配的距离，包括从给定位置从前查找和从后查找

for (int xi = Math.max(mi - range, 0), xn = Math.min(mi + range + 1, max

.length()); xi < xn; xi++) {

// 排除被匹配过的字符，若找到匹配的字符，则停止

if (!matchFlags[xi] && c1 == max.charAt(xi)) {

matchIndexes[mi] = xi;

matchFlags[xi] = true;

matches++;

break;

}

}

}

// 记录min字符串里匹配的字符串，保持顺序

char[] ms1 = new char[matches];

// 记录max字符串里匹配的字符串，保持顺序

char[] ms2 = new char[matches];

for (int i = 0, si = 0; i < min.length(); i++) {

if (matchIndexes[i] != -1) {

ms1[si] = min.charAt(i);

si++;

}

}

for (int i = 0, si = 0; i < max.length(); i++) {

if (matchFlags[i]) {

ms2[si] = max.charAt(i);

si++;

}

}

// 查找换位的数目

int transpositions = 0;

for (int mi = 0; mi < ms1.length; mi++) {

if (ms1[mi] != ms2[mi]) {

transpositions++;

}

}

// 查找相同前缀的数目

int prefix = 0;

for (int mi = 0; mi < min.length(); mi++) {

if (s1.charAt(mi) == s2.charAt(mi)) {

prefix++;

} else {

break;

}

}

// 返回匹配数目（m），换位的数目（t），相同的前缀的数目，字符串最长

return new int[] { matches, transpositions / 2, prefix, max.length() };

}

public float getDistance(String s1, String s2) {

int[] mtp = matches(s1, s2);

// 返回匹配数目（m）

float m = (float) mtp[0];

if (m == 0) {

return 0f;

}

// Jaro Distance

float j = ((m / s1.length() + m / s2.length() + (m - mtp[1]) / m)) / 3;

// 计算Jaro-Winkler Distance， 这里调整分数的因数=Math.min(0.1f, 1f / mtp[3])

float jw = j < getThreshold() ? j : j + Math.min(0.1f, 1f / mtp[3]) \* mtp[2]

\* (1 - j);

return jw;

}

/\*\*

\* Sets the threshold used to determine when Winkler bonus should be used.

\* Set to a negative value to get the Jaro distance.

\* @param threshold the new value of the threshold

\*/

public void setThreshold(float threshold) {

this.threshold = threshold;

}

/\*\*

\* Returns the current value of the threshold used for adding the Winkler bonus.

\* The default value is 0.7.

\* @return the current value of the threshold

\*/

public float getThreshold() {

return threshold;

}

}

编辑距离概念描述：

编辑距离，又称Levenshtein距离，是指两个字串之间，由一个转成另一个所需的最少编辑操作次数。许可的编辑操作包括将一个字符替换成另一个字符，插入一个字符，删除一个字符。

例如将kitten一字转成sitting：

sitten （k→s）

sittin （e→i）

sitting （→g）

俄罗斯科学家Vladimir Levenshtein在1965年提出这个概念。

问题：找出字符串的编辑距离，即把一个字符串s1最少经过多少步操作变成编程字符串s2，操作有三种，添加一个字符，删除一个字符，修改一个字符

解析：

首先定义这样一个函数——edit(i, j)，它表示第一个字符串的长度为i的子串到第二个字符串的长度为j的子串的编辑距离。

显然可以有如下动态规划公式：

<!--[if !supportLists]-->if i == 0 且 j == 0，edit(i, j) = 0

<!--[if !supportLists]-->if i == 0 且 j > 0，edit(i, j) = j

<!--[if !supportLists]-->if i > 0 且j == 0，edit(i, j) = i

<!--[if !supportLists]-->if i ≥ 1 且 j ≥ 1 ，edit(i, j) == min{ edit(i-1, j) + 1, edit(i, j-1) + 1, edit(i-1, j-1) + f(i, j) }，当第一个字符串的第i个字符不等于第二个字符串的第j个字符时，f(i, j) = 1；否则，f(i, j) = 0。

0

f

a

i

l

i

n

g

0

s

a

i

l

n

0

f

a

i

l

i

n

g

0

0

1

2

3

4

5

6

7

s

1

a

2

i

3

l

4

n

5

计算edit(1, 1)，edit(0, 1) + 1 == 2，edit(1, 0) + 1 == 2，edit(0, 0) + f(1, 1) == 0 + 1 == 1，min(edit(0, 1)，edit(1, 0)，edit(0, 0) + f(1, 1))==1，因此edit(1, 1) == 1。 依次类推：

0

f

a

i

l

i

n

g

0

0

1

2

3

4

5

6

7

s

1

1

2

3

4

5

6

7

a

2

2

i

3

l

4

n

5

edit(2, 1) + 1 == 3，edit(1, 2) + 1 == 3，edit(1, 1) + f(2, 2) == 1 + 0 == 1，其中s1[2] == 'a' 而 s2[1] == 'f'‘，两者不相同，所以交换相邻字符的操作不计入比较最小数中计算。以此计算，得出最后矩阵为：

0

f

a

i

l

i

n

g

0

0

1

2

3

4

5

6

7

s

1

1

2

3

4

5

6

7

a

2

2

1

2

3

4

5

6

i

3

3

2

1

2

3

4

5

l

4

4

3

2

1

2

3

4

n

5

5

4

3

2

2

2

3

Lucene里的实现：

Java代码 收藏代码

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Compute Levenshtein distance: see org.apache.commons.lang.StringUtils#getLevenshteinDistance(String, String)

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public float getDistance (String target, String other) {

char[] sa;

int n;

int p[]; //'previous' cost array, horizontally

int d[]; // cost array, horizontally

int \_d[]; //placeholder to assist in swapping p and d

/\*

\* 这里的实现是仅仅使用一个临时数组做交换空间，避免直接创建

\* 一个二维数组可能出现的oom

\*/

sa = target.toCharArray();

n = sa.length;

// 初始化目标字符串

p = new int[n+1];

d = new int[n+1];

final int m = other.length();

if (n == 0 || m == 0) {

if (n == m) {

return 1;

}

else {

return 0;

}

}

// indexes into strings s and t

int i; // iterates through s

int j; // iterates through t

char t\_j; // jth character of t

int cost; // cost

// 横向字符串，初始化编辑距离 0-n, 第一个横排的值

for (i = 0; i<=n; i++) {

p[i] = i;

}

for (j = 1; j<=m; j++) { // 竖向字符串，从第二个横排开始

// 注意，这里j-1是由于数组的第一个位置是0，字符串位置从1开始的，实际取字符时，要位置-1得到下标

t\_j = other.charAt(j-1);

// 横向编辑距离的第一个值的值初始值（1-m)

d[0] = j;

for (i=1; i<=n; i++) {

cost = sa[i-1]==t\_j ? 0 : 1;

// edit(i, j) == min{ edit(i-1, j) + 1, edit(i, j-1) + 1, edit(i-1, j-1) + f(i, j) }

// minimum of cell to the left+1, to the top+1, diagonally left and up +cost

d[i] = Math.min(Math.min(d[i-1]+1, p[i]+1), p[i-1]+cost);

}

// copy current distance counts to 'previous row' distance counts

\_d = p;

// 记录上一个横排的值

p = d;

// 下一个马上处理的横排

d = \_d;

}

// our last action in the above loop was to switch d and p, so p now

// actually has the most recent cost counts

return 1.0f - ((float) p[n] / Math.max(other.length(), sa.length));

}

就是N-Gram version of edit distance

Java代码 收藏代码

public float getDistance(String source, String target) {

final int sl = source.length();

final int tl = target.length();

if (sl == 0 || tl == 0) {

if (sl == tl) {

return 1;

}

else {

return 0;

}

}

int cost = 0;

if (sl < n || tl < n) {

for (int i=0,ni=Math.min(sl,tl);i<ni;i++) {

if (source.charAt(i) == target.charAt(i)) {

cost++;

}

}

return (float) cost/Math.max(sl, tl);

}

char[] sa = new char[sl+n-1];

float p[]; //'previous' cost array, horizontally

float d[]; // cost array, horizontally

float \_d[]; //placeholder to assist in swapping p and d

//construct sa with prefix

// 填充前缀，满足n-gram

for (int i=0;i<sa.length;i++) {

if (i < n-1) {

sa[i]=0; //add prefix

}

else {

sa[i] = source.charAt(i-n+1);

}

}

p = new float[sl+1];

d = new float[sl+1];

// indexes into strings s and t

int i; // iterates through source

int j; // iterates through target

char[] t\_j = new char[n]; // jth n-gram of t

// 初始化第一横排的编辑距离

for (i = 0; i<=sl; i++) {

p[i] = i;

}

for (j = 1; j<=tl; j++) { // 开始处理第二个横排，...到tl最后一个横排

//construct t\_j n-gram，构建n-gram

if (j < n) { // 补充前缀

for (int ti=0;ti<n-j;ti++) {

t\_j[ti]=0; //add prefix

}

for (int ti=n-j;ti<n;ti++) {

t\_j[ti]=target.charAt(ti-(n-j));

}

}

else { // 直接取n-gram

t\_j = target.substring(j-n, j).toCharArray();

}

d[0] = j;

for (i=1; i<=sl; i++) {

cost = 0;

int tn=n;

//compare sa to t\_j，计算f(i,j)

for (int ni=0;ni<n;ni++) {

if (sa[i-1+ni] != t\_j[ni]) {

cost++;

}

else if (sa[i-1+ni] == 0) { //discount matches on prefix

tn--;

}

}

float ec = (float) cost/tn;

// minimum of cell to the left+1, to the top+1, diagonally left and up +cost

d[i] = Math.min(Math.min(d[i-1]+1, p[i]+1), p[i-1]+ec);

}

// copy current distance counts to 'previous row' distance counts

\_d = p;

p = d;

d = \_d;

}

// our last action in the above loop was to switch d and p, so p now

// actually has the most recent cost counts

return 1.0f - ((float) p[sl] / Math.max(tl, sl));

}

Approximate string matching

Bitap algorithm

Damerau–Levenshtein distance

Edit distance

Hamming distance

Jaro–Winkler distance

Lee distance

Levenshtein automaton

Levenshtein distance

Wagner–Fischer algorithm