

走在前往架构师的路上

专注于分布式计算，大数据，数据挖掘，机器学习算法等领域的研究

目录视图

摘要视图

RSS 订阅

Follow me

<https://github.com/linyiqun>

个人资料



Android路上的人

关注

发私信

访问：1331252次

积分：12637

等级：BLOG > ?

排名：第1294名

原创：266篇

转载：0篇

译文：2篇

评论：306条

博主介绍

Apache Hadoop Committe
r，其中主要专注于HDFS模
块。毕业于HDU计算机系，研究
领域分布式存储。曾就职于国内
女性电商平台蘑菇街，目前就职
于唯品会上海研发中心，数据平
台与应用部门。

新书发布

PrefixSpan序列模式挖掘算法

标签：数据挖掘 算法 机器学习

2015-02-12 19:0613053人阅读评论(7)收藏举报

分类：

机器学习（30）数据挖掘（32）算法（44）

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录(?)

[+]

更多数据挖掘代码：<https://github.com/linyiqun/DataMiningAlgorithm>

介绍

与GSP一样，PrefixSpan算法也是序列模式分析算法的一种，不过与前者不同的是PrefixSpan算法不产生任何的候选集，在这点上可以说已经比GSP好很多了。PrefixSpan算法可以挖掘出满足阈值的所有序列模式，可以说是非常经典的算法。序列的格式就是上文中提到过的类似于<a, b, (de)>这样的。

算法原理

PrefixSpan算法的原理是采用后缀序列转前缀序列的方式来构造频繁序列的。举个例子，

abc	abc	ac	d	cf
ad	c	abcd	ae	
ef	ab	df	c	b
e	g	af	c	b
			c	

比如原始序列如上图所示，4条序列，1个序列中好几个项集，项集内有1个或多个元素，首先找出前缀为a的子序列，此时序列前缀为<a>,后缀就变为了：

a				
_bc	abc	ac	d	cf
_d	c	abcd	ae	
_b	df	c	b	
_f	c	b	c	

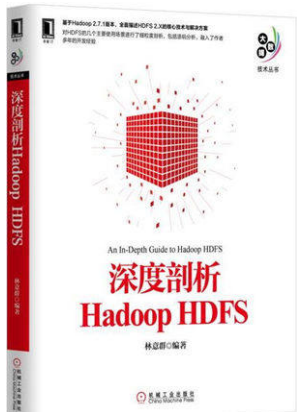
"_"下标符代表前缀为a，说明是在项集中匹配的。这就相当于从后缀序列中提取出1项加入到前缀序列中，变化的规则就是从左往右扫描，找到第1个此元素对应的项，然后做改变。然后根据此规则继续递归直到后续序列不满足最小支持度阈值的情况。所以此算法的难点就转变为了从后缀序列变为前缀序列的过程。在这个过程要分为2种情况，第1种是单个元素项的后缀提前，比如这里的a，对单个项的提前有分为几种情况，比如：

<b a c ad>，就会变为<c ad>，如果a是嵌套在项集中的情况<b c dad r>，就会变为<_d r>，_代
表的的就是a.如果a在一项的最末尾，此项也会被移除<b c dda r>变为<r>。但是如果是这种情况<_da d
d>a包含在下标符中，将会做处理，应该此时的a是在前缀序列所属的项集内的。

还有1个大类的分类就是对于组合项的后缀提取，可以分为2个情况，1个是从_X中寻找，一个从后面找出连
续的项集，比如在这里<a>的条件下，找出前缀<(ab)>的后缀序列

<(a b) > :3				
_c	abc	ac	d	cf
_cd	ae			
df	c	b		

第一种在_X中寻找还有没有X=a的情况，因为_已经代表1个a了，还有一个是判断_X != _a的情况，从后
面的项集中找到包含有连续的aa的那个项集，然后做变换处理，与单个项集的变换规则一致。



新书<<深度剖析Hadoop HDFS>>发布上市，此书源自于笔者博客，重新经过整理，完善而成，此书的定位并不是一本纯源码分析的书籍，其中有许多笔者在工作和学习中对于HDFS的一些有趣的看法和理解。链接：

淘宝
京东

博客专栏



经典数据挖掘算法

文章：29篇
阅读：205420



HDFS源码分析

文章：6篇
阅读：17732



Hadoop Common源码学习

文章：8篇
阅读：21417



MapReduce源码分析

文章：11篇
阅读：28773



Redis源码分析

文章：36篇
阅读：182840

文章搜索

文章分类

- Jvm (2)
- BigData (13)
- Nosql (3)
- Java 中间件 (1)
- 设计模式 (20)
- 算法 (45)

算法的递归顺序

想要实现整个的序列挖掘，算法的递归顺序就显得非常重要了。在探索递归顺序的路上还是犯了一些错误的，刚开始的递归顺序是<a>----><a a>----><a a a>，假设<a a a>找不到对应的后缀模式时，然后回溯到<a (aa)>进行递归，后来发现这样会漏掉情况，为什么呢，因为如果<a a>没法进行到<a a a>，那么就不可能会有前缀<a (aa)>，顶多会判断到<(aa)>，从<a a>处回调的。于是我发现了这个问题，就变为了下面这个样子，经测试是对的。：

加入所有的单个元素的类似为a-f,顺序为

<a>，---><a a>，同时<(aa)>，然后<ab>同时<(ab)>，就是在a添加a-f的元素的时候，检验a所属项集添加a-f元素的情况。这样就不会漏掉情况了，用了2个递归搞定了这个问题。这个算法的整体实现可以对照代码来看会理解很多。最后提醒一点，在每次做出改变之后都会判断一下是否满足最小支持度阈值的。

PrefixSpan实例

这里举1个真实一点的例子，下面是输入的初始序列：

<(b d) c b (a c)>
<(b f) (c e) b (f g)>
<(a h) (b f) a b f>
<(b e) (c e) d>
<a (b d) b c b (a d e)>

挖掘出的所有的序列模式为，下面是一个表格的形式

Prefix	Projected (postfix) database	Sequential patterns
<a>	<(_c)> <(bf)abf> <(bd)bc b(ade)>	<a>, <aa>, <ab>, <aba>, <abb>
	<(_d)cb(ac)> <(_f)(ce)bf> <(_f)abf> <(_e)(ce)d> <(_d)bc b(ade)>	, <ba>, <bb>, <bba>, <bbc>, <bfb>, <bc>, <bca>, <bcb>, <bcb a>, <bcd>, <b(ce)>, <bd>, <(bd)>, <(bd)a>, <(bd)b>, <(bd)ba>, <(bd)bc>, <(bd)c>, <(bd)ca>, <(bd)cb>, <(bd)cba>, <be>, <bf>, <(bf)>, <(bf)b>, <(bf)bf>, <(bf)f>
<c>	<b(ac)> <(_e)bf> <(_e)d> <b(ade)>	<c>, <ca>, <cb>, <cd>, <(ce)>, <cba>
<d>	<cb(ac)> <bcb(ade)>	<d>, <da>, <db>, <dc>, <d b a>, <d b c>, <dca>, <dc b>
<e>	<bf>	<e>
<f>	<(ce)bf> <abf>	<f>, <fb>, <ff>, <fbf>

在的序列模式中少了1个序列模式。可以与后面程序算法测试的结果做对比。

算法的代码实现

代码实现同样以这个为例子，这样会显得更有说服力。

测试数据：

```
[java]
01. bd c b ac
02. bf ce b fg
03. ah bf a b f
04. be ce d
05. a bd b c b ade
```

Sequence.java:

```
[java]
01. package DataMining_PrefixSpan;
```

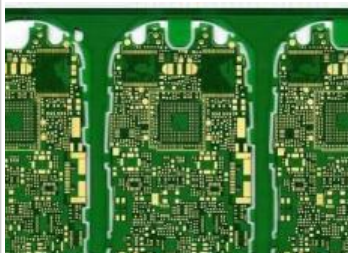
数据结构	(3)
Linux Kernel	(2)
JDK源码	(7)
安全	(5)
Redis源码	(36)
Storm	(6)
Spark	(1)
Hadoop	(115)
MapReduce	(11)
HDFS	(75)
YARN	(15)
Hadoop Common	(9)
数据挖掘	(33)
机器学习	(31)
分布式系统	(38)
搜索引擎	(6)
系统运维	(17)

文章存档	
2018年01月	(1)
2017年12月	(2)
2017年11月	(2)
2017年10月	(2)
2017年09月	(2)
展开	

阅读排行	
从Container内存监控限制到C...	(283770)
Redis源码分析（一）--Redis...	(32302)
决策分类树算法之ID3，C4.5...	(17703)
HDFS内存存储	(16179)
HDFS副本放置策略	(14963)
HDFS数据加密空间--Encrypt...	(14461)
Redis源码分析（三十六）--- ...	(14446)
Apriori算法--关联规则挖掘	(13496)
PrefixSpan序列模式挖掘算法	(13042)
Hadoop节点“慢磁盘”监控	(12982)

评论排行	
qSpan频繁子图挖掘算法	(24)
决策分类树算法之ID3，C4.5...	(16)
18大经典数据挖掘算法小结	(15)
Redis源码分析（三）---dict...	(13)
Redis源码分析（二）--结构...	(12)
记一次DataNode慢启动问题	(10)
Redis源码分析（一）--Redis...	(10)
Hadoop Balancer源码解读	(8)
PrefixSpan序列模式挖掘算法	(7)
朴素贝叶斯分类算法	(7)

```
02.
03. import java.util.ArrayList;
04.
05. /**
06.  * 序列类
07.  *
08.  * @author lyq
09.  *
10.  */
11. public class Sequence {
12.     // 序列内的项集
13.     private ArrayList<ItemSet> itemSetList;
14.
15.     public Sequence() {
16.         this.itemSetList = new ArrayList<>();
17.     }
18.
19.     public ArrayList<ItemSet> getItemSetList() {
20.         return itemSetList;
21.     }
22.
23.     public void setItemSetList(ArrayList<ItemSet> itemSetList) {
24.         this.itemSetList = itemSetList;
25.     }
26.
27.     /**
28.      * 判断单一项是否包含于此序列
29.      *
30.      * @param c
31.      *      待判断项
32.      * @return
33.      */
34.     public boolean strIsContained(String c) {
35.         boolean isContained = false;
36.
37.         for (ItemSet itemSet : itemSetList) {
38.             isContained = false;
39.
40.             for (String s : itemSet.getItems()) {
41.                 if (itemSet.getItems().contains("_")) {
42.                     continue;
43.                 }
44.
45.                 if (s.equals(c)) {
46.                     isContained = true;
47.                     break;
48.                 }
49.             }
50.
51.             if (isContained) {
52.                 // 如果已经检测出包含了，直接跳出循环
53.                 break;
54.             }
55.         }
56.
57.         return isContained;
58.     }
59.
60.     /**
61.      * 判断组合项集是否包含于序列中
62.      *
63.      * @param itemSet
64.      *      组合的项集，元素超过1个
65.      * @return
66.      */
67.     public boolean compoentItemIsContain(ItemSet itemSet) {
68.         boolean isContained = false;
69.         ArrayList<String> tempItems;
70.         String lastItem = itemSet.getLastValue();
71.
72.         for (int i = 0; i < this.itemSetList.size(); i++) {
73.             tempItems = this.itemSetList.get(i).getItems();
74.             // 分2种情况查找，第一种从_x中找出x等于项集最后的元素，因为_前缀已经为原本的元素
75.             if (tempItems.size() > 1 && tempItems.get(0).equals("_")
76.                 && tempItems.get(1).equals(lastItem)) {
77.                 isContained = true;
78.                 break;
79.             } else if (!tempItems.get(0).equals("_")) {
80.                 // 从没有_前缀的项集开始寻找，第二种为从后面的后缀中找出直接找出连续字符为ab为同一项集的项集
81.                 if (strArrayContains(tempItems, itemSet.getItems())) {
```



电路板抄板



最新评论

HDFS镜像文件的解析与反解析

jhonsnjs : 大佬, 大佬

HDFS数据加密空间--Encryption zone

WaitMeFun : 如果不开启kerberos, 对文件进行加解密的方式可以达到非法下载查看内容, 但是无法解决不断上传, ...

Redis源码分析 (一) --Redis结构解析

coderlong : @limingji0503:现在已经没有乱码了, 兄弟。



智能停车场



HDFS数据资源使用量分析以及趋势预测

呆丶 : 楼主, 我想问一下, 如何才能获取到HDFS中block的访问频率呢

Redis源码解析 (十五) --- aof-append ...

永志 : 博主说的有点问题, 不是内存满了才刷到磁盘上的

NameNode处理上报block块逻辑分析

hunanmeitan : @Androidlushangdere n:因为交换机故障, 导致几十个DN和NN通信中断。当网络恢复的...

```

82.         isContained = true;
83.         break;
84.     }
85. }
86.
87.     if (isContained) {
88.         break;
89.     }
90. }
91.
92.     return isContained;
93. }
94.
95. /**
96.  * 删除单个项
97.  *
98.  * @param s
99.  *      待删除项
100.  */
101. public void deleteSingleItem(String s) {
102.     ArrayList<String> tempItems;
103.     ArrayList<String> deleteItems = new ArrayList<>();
104.
105.     for (ItemSet itemSet : this.itemSetList) {
106.         tempItems = itemSet.getItems();
107.         deleteItems = new ArrayList<>();
108.
109.         for (int i = 0; i < tempItems.size(); i++) {
110.             if (tempItems.get(i).equals(s)) {
111.                 deleteItems.add(tempItems.get(i));
112.             }
113.         }
114.
115.         tempItems.removeAll(deleteItems);
116.     }
117. }
118.
119. /**
120.  * 提取项s之后所得的序列
121.  *
122.  * @param s
123.  *      目标提取项s
124.  */
125. public Sequence extractItem(String s) {
126.     Sequence extractSeq = this.copySequence();
127.     ItemSet itemSet;
128.     ArrayList<String> items;
129.     ArrayList<ItemSet> deleteItemSets = new ArrayList<>();
130.     ArrayList<String> tempItems = new ArrayList<>();
131.
132.     for (int k = 0; k < extractSeq.itemSetList.size(); k++) {
133.         itemSet = extractSeq.itemSetList.get(k);
134.         items = itemSet.getItems();
135.         if (items.size() == 1 && items.get(0).equals(s)) {
136.             //如果找到的是单项, 则完全移除, 跳出循环
137.             extractSeq.itemSetList.remove(k);
138.             break;
139.         } else if (items.size() > 1 && !items.get(0).equals("_")) {
140.             //在后续的多元素项中判断是否包含此元素
141.             if (items.contains(s)) {
142.                 //如果包含把s后面的元素加入到临时字符串数组中
143.                 int index = items.indexOf(s);
144.                 for (int j = index; j < items.size(); j++) {
145.                     tempItems.add(items.get(j));
146.                 }
147.                 //将第一位的s变成下标符 "_"
148.                 tempItems.set(0, "_");
149.                 if (tempItems.size() == 1) {
150.                     // 如果此匹配为在最末端, 同样移除
151.                     deleteItemSets.add(itemSet);
152.                 } else {
153.                     //将变化后的项替换原来的
154.                     extractSeq.itemSetList.set(k, new ItemSet(tempItems));
155.                 }
156.                 break;
157.             } else {
158.                 deleteItemSets.add(itemSet);
159.             }
160.         } else {
161.             // 不符合以上2项条件的统统移除
162.             deleteItemSets.add(itemSet);

```

```

163.         }
164.     }
165.     extractSeq.itemSetList.removeAll(deleteItemSets);
166.
167.     return extractSeq;
168. }
169.
170. /**
171.  * 提取组合项之后的序列
172.  *
173.  * @param array
174.  *      组合数组
175.  * @return
176.  */
177. public Sequence extractComponentItem(ArrayList<String> array) {
178.     // 找到目标项, 是否立刻停止
179.     boolean stopExtract = false;
180.     Sequence seq = this.copySequence();
181.     String lastItem = array.get(array.size() - 1);
182.     ArrayList<String> tempItems;
183.     ArrayList<ItemSet> deleteItems = new ArrayList<>();
184.
185.     for (int i = 0; i < seq.itemSetList.size(); i++) {
186.         if (stopExtract) {
187.             break;
188.         }
189.
190.         tempItems = seq.itemSetList.get(i).getItemSets();
191.         // 分2种情况查找, 第一种从_x中找出x等于项集最后的元素, 因为_前缀已经为本来的元素
192.         if (tempItems.size() > 1 && tempItems.get(0).equals("_")
193.             && tempItems.get(1).equals(lastItem)) {
194.             if (tempItems.size() == 2) {
195.                 seq.itemSetList.remove(i);
196.             } else {
197.                 // 把1号位置变为下标符 "_", 往后移1个字符的位置
198.                 tempItems.set(1, "_");
199.                 // 移除第一个的 "_" 下划符
200.                 tempItems.remove(0);
201.             }
202.             stopExtract = true;
203.             break;
204.         } else if (!tempItems.get(0).equals("_")) {
205.             // 从没有_前缀的项集开始寻找, 第二种为从后面的后缀中找出直接找出连续字符为ab为同
206.             // 一项集的项集
207.             if (strArrayContains(tempItems, array)) {
208.                 // 从左往右找出第一个给定字符的位置, 把后面的部分截取出来
209.                 int index = tempItems.indexOf(lastItem);
210.                 ArrayList<String> array2 = new ArrayList<String>();
211.
212.                 for (int j = index; j < tempItems.size(); j++) {
213.                     array2.add(tempItems.get(j));
214.                 }
215.                 array2.set(0, "_");
216.
217.                 if (array2.size() == 1) {
218.                     // 如果此项在末尾的位置, 则移除该项, 否则进行替换
219.                     deleteItems.add(seq.itemSetList.get(i));
220.                 } else {
221.                     seq.itemSetList.set(i, new ItemSet(array2));
222.                 }
223.                 stopExtract = true;
224.                 break;
225.             } else {
226.                 deleteItems.add(seq.itemSetList.get(i));
227.             }
228.         } else {
229.             // 这种情况是处理_x中x不等于最后一个元素的情况
230.             deleteItems.add(seq.itemSetList.get(i));
231.         }
232.     }
233.
234.     seq.itemSetList.removeAll(deleteItems);
235.
236.     return seq;
237. }
238.
239. /**
240.  * 深拷贝一个序列
241.  *
242.  * @return
243.  */

```

```

243.     public Sequence copySequence() {
244.         Sequence copySeq = new Sequence();
245.         ItemSet tempItemSet;
246.         ArrayList<String> items;
247.
248.         for (ItemSet itemSet : this.itemSetList) {
249.             items = (ArrayList<String>) itemSet.getItems().clone();
250.             tempItemSet = new ItemSet(items);
251.             copySeq.getItemSetList().add(tempItemSet);
252.         }
253.
254.         return copySeq;
255.     }
256.
257.     /**
258.      * 获取序列中最后一个项集的最后1个元素
259.      *
260.      * @return
261.      */
262.     public String getLastItemSetValue() {
263.         int size = this.getItemSetList().size();
264.         ItemSet itemSet = this.getItemSetList().get(size - 1);
265.         size = itemSet.getItems().size();
266.
267.         return itemSet.getItems().get(size - 1);
268.     }
269.
270.     /**
271.      * 判断strList2是否是strList1的子序列
272.      *
273.      * @param strList1
274.      * @param strList2
275.      * @return
276.      */
277.     public boolean strArrayContains(ArrayList<String> strList1,
278.                                     ArrayList<String> strList2) {
279.         boolean isContained = false;
280.
281.         for (int i = 0; i < strList1.size() - strList2.size() + 1; i++) {
282.             isContained = true;
283.
284.             for (int j = 0, k = i; j < strList2.size(); j++, k++) {
285.                 if (!strList1.get(k).equals(strList2.get(j))) {
286.                     isContained = false;
287.                     break;
288.                 }
289.             }
290.
291.             if (isContained) {
292.                 break;
293.             }
294.         }
295.
296.         return isContained;
297.     }
298. }

```

ItemSet.java :

```

[java]
01. package DataMining_PrefixSpan;
02.
03. import java.util.ArrayList;
04.
05. /**
06.  * 字符项集类
07.  *
08.  * @author lyq
09.  */
10.
11. public class ItemSet {
12.     // 项集内的字符
13.     private ArrayList<String> items;
14.
15.     public ItemSet(String[] str) {
16.         items = new ArrayList<>();
17.         for (String s : str) {
18.             items.add(s);
19.         }

```



```

20.     }
21.
22.     public ItemSet(ArrayList<String> itemsList) {
23.         this.items = itemsList;
24.     }
25.
26.     public ItemSet(String s) {
27.         items = new ArrayList<>();
28.         for (int i = 0; i < s.length(); i++) {
29.             items.add(s.charAt(i) + "");
30.         }
31.     }
32.
33.     public ArrayList<String> getItems() {
34.         return items;
35.     }
36.
37.     public void setItems(ArrayList<String> items) {
38.         this.items = items;
39.     }
40.
41.     /**
42.      * 获取项集最后1个元素
43.      *
44.      * @return
45.      */
46.     public String getLastValue() {
47.         int size = this.items.size();
48.
49.         return this.items.get(size - 1);
50.     }
51. }

```

PrefixSpanTool.java :

```

[java]
01. package DataMining_PrefixSpan;
02.
03. import java.io.BufferedReader;
04. import java.io.File;
05. import java.io.FileReader;
06. import java.io.IOException;
07. import java.util.ArrayList;
08. import java.util.Collections;
09. import java.util.HashMap;
10. import java.util.Map;
11.
12. /**
13.  * PrefixSpanTool序列模式分析算法工具类
14.  *
15.  * @author lyq
16.  *
17.  */
18. public class PrefixSpanTool {
19.     // 测试数据文件地址
20.     private String filePath;
21.     // 最小支持度阈值比例
22.     private double minSupportRate;
23.     // 最小支持度, 通过序列总数乘以阈值比例计算
24.     private int minSupport;
25.     // 原始序列组
26.     private ArrayList<Sequence> totalSeqs;
27.     // 挖掘出的所有序列频繁模式
28.     private ArrayList<Sequence> totalFrequentSeqs;
29.     // 所有的单一项, 用于递归枚举
30.     private ArrayList<String> singleItems;
31.
32.     public PrefixSpanTool(String filePath, double minSupportRate) {
33.         this.filePath = filePath;
34.         this.minSupportRate = minSupportRate;
35.         readDataFile();
36.     }
37.
38.     /**
39.      * 从文件中读取数据
40.      */
41.     private void readDataFile() {
42.         File file = new File(filePath);
43.         ArrayList<String[]> dataArray = new ArrayList<String[]>();

```

```

44.
45.     try {
46.         BufferedReader in = new BufferedReader(new FileReader(file));
47.         String str;
48.         String[] tempArray;
49.         while ((str = in.readLine()) != null) {
50.             tempArray = str.split(" ");
51.             dataArray.add(tempArray);
52.         }
53.         in.close();
54.     } catch (IOException e) {
55.         e.printStackTrace();
56.     }
57.
58.     minSupport = (int) (dataArray.size() * minSupportRate);
59.     totalSeqs = new ArrayList<>();
60.     totalFrequentSeqs = new ArrayList<>();
61.     Sequence tempSeq;
62.     ItemSet tempItemSet;
63.     for (String[] str : dataArray) {
64.         tempSeq = new Sequence();
65.         for (String s : str) {
66.             tempItemSet = new ItemSet(s);
67.             tempSeq.getItemSetList().add(tempItemSet);
68.         }
69.         totalSeqs.add(tempSeq);
70.     }
71.
72.     System.out.println("原始序列数据: ");
73.     outputSequence(totalSeqs);
74. }
75.
76. /**
77.  * 输出序列列表内容
78.  *
79.  * @param seqList
80.  *      待输出序列列表
81.  */
82. private void outputSequence(ArrayList<Sequence> seqList) {
83.     for (Sequence seq : seqList) {
84.         System.out.print("<");
85.         for (ItemSet itemSet : seq.getItemSetList()) {
86.             if (itemSet.getItems().size() > 1) {
87.                 System.out.print("(");
88.             }
89.
90.             for (String s : itemSet.getItems()) {
91.                 System.out.print(s + " ");
92.             }
93.
94.             if (itemSet.getItems().size() > 1) {
95.                 System.out.print(")");
96.             }
97.         }
98.         System.out.println(">");
99.     }
100. }
101.
102. /**
103.  * 移除初始序列中不满足最小支持度阈值的单项
104.  */
105. private void removeInitSeqsItem() {
106.     int count = 0;
107.     HashMap<String, Integer> itemMap = new HashMap<>();
108.     singleItems = new ArrayList<>();
109.
110.     for (Sequence seq : totalSeqs) {
111.         for (ItemSet itemSet : seq.getItemSetList()) {
112.             for (String s : itemSet.getItems()) {
113.                 if (!itemMap.containsKey(s)) {
114.                     itemMap.put(s, 1);
115.                 }
116.             }
117.         }
118.     }
119.
120.     String key;
121.     for (Map.Entry entry : itemMap.entrySet()) {
122.         count = 0;
123.         key = (String) entry.getKey();
124.         for (Sequence seq : totalSeqs) {

```



```

125.         if (seq.strIsContained(key)) {
126.             count++;
127.         }
128.     }
129.
130.     itemMap.put(key, count);
131.
132. }
133.
134. for (Map.Entry entry : itemMap.entrySet()) {
135.     key = (String) entry.getKey();
136.     count = (int) entry.getValue();
137.
138.     if (count < minSupport) {
139.         // 如果支持度阈值小于所得的最小支持度阈值, 则删除该项
140.         for (Sequence seq : totalSeqs) {
141.             seq.deleteSingleItem(key);
142.         }
143.     } else {
144.         singleItems.add(key);
145.     }
146. }
147.
148. Collections.sort(singleItems);
149. }
150.
151. /**
152.  * 递归搜索满足条件的序列模式
153.  *
154.  * @param beforeSeq
155.  *      前缀序列
156.  * @param afterSeqList
157.  *      后缀序列列表
158.  */
159. private void recursiveSearchSeqs(Sequence beforeSeq,
160.     ArrayList<Sequence> afterSeqList) {
161.     ItemSet tempItemSet;
162.     Sequence tempSeq2;
163.     Sequence tempSeq;
164.     ArrayList<Sequence> tempSeqList = new ArrayList<>();
165.
166.     for (String s : singleItems) {
167.         // 分成2种形式递归, 以<a>为起始项, 第一种直接加入独立项集遍历<a,a>,<a,b> <a,c>...
168.         if (isLargerThanMinSupport(s, afterSeqList)) {
169.             tempSeq = beforeSeq.copySequence();
170.             tempItemSet = new ItemSet(s);
171.             tempSeq.getItemSetList().add(tempItemSet);
172.
173.             totalFrequentSeqs.add(tempSeq);
174.
175.             tempSeqList = new ArrayList<>();
176.             for (Sequence seq : afterSeqList) {
177.                 if (seq.strIsContained(s)) {
178.                     tempSeq2 = seq.extractItem(s);
179.                     tempSeqList.add(tempSeq2);
180.                 }
181.             }
182.
183.             recursiveSearchSeqs(tempSeq, tempSeqList);
184.         }
185.
186.         // 第二种递归为以元素的身份加入最后的项集内以a为例<aa>,<(ab)>,<(ac)>...
187.         // a在这里可以理解为一个前缀序列, 里面可能是单个元素或者已经是多元素的项集
188.         tempSeq = beforeSeq.copySequence();
189.         int size = tempSeq.getItemSetList().size();
190.         tempItemSet = tempSeq.getItemSetList().get(size - 1);
191.         tempItemSet.getItems().add(s);
192.
193.         if (isLargerThanMinSupport(tempItemSet, afterSeqList)) {
194.             tempSeqList = new ArrayList<>();
195.             for (Sequence seq : afterSeqList) {
196.                 if (seq.compoentItemIsContain(tempItemSet)) {
197.                     tempSeq2 = seq.extractCompoentItem(tempItemSet
198.                         .getItems());
199.                     tempSeqList.add(tempSeq2);
200.                 }
201.             }
202.             totalFrequentSeqs.add(tempSeq);
203.
204.             recursiveSearchSeqs(tempSeq, tempSeqList);
205.         }

```

```

206.     }
207. }
208.
209. /**
210.  * 所传入的项组合在所给定序列中的支持度是否超过阈值
211.  *
212.  * @param s
213.  *      所需匹配的项
214.  * @param seqList
215.  *      比较序列数据
216.  * @return
217.  */
218. private boolean isLargerThanMinSupport(String s, ArrayList<Sequence> seqList) {
219.     boolean isLarge = false;
220.     int count = 0;
221.
222.     for (Sequence seq : seqList) {
223.         if (seq.strIsContained(s)) {
224.             count++;
225.         }
226.     }
227.
228.     if (count >= minSupport) {
229.         isLarge = true;
230.     }
231.
232.     return isLarge;
233. }
234.
235. /**
236.  * 所传入的组合项集在序列中的支持度是否大于阈值
237.  *
238.  * @param itemSet
239.  *      组合元素项集
240.  * @param seqList
241.  *      比较的序列列表
242.  * @return
243.  */
244. private boolean isLargerThanMinSupport(ItemSet itemSet,
245.     ArrayList<Sequence> seqList) {
246.     boolean isLarge = false;
247.     int count = 0;
248.
249.     if (seqList == null) {
250.         return false;
251.     }
252.
253.     for (Sequence seq : seqList) {
254.         if (seq.compoentItemIsContain(itemSet)) {
255.             count++;
256.         }
257.     }
258.
259.     if (count >= minSupport) {
260.         isLarge = true;
261.     }
262.
263.     return isLarge;
264. }
265.
266. /**
267.  * 序列模式分析计算
268.  */
269. public void prefixSpanCalculate() {
270.     Sequence seq;
271.     Sequence tempSeq;
272.     ArrayList<Sequence> tempSeqList = new ArrayList<>();
273.     ItemSet itemSet;
274.     removeInitSeqsItem();
275.
276.     for (String s : singleItems) {
277.         // 从最开始的a,b,d开始递归往下寻找频繁序列模式
278.         seq = new Sequence();
279.         itemSet = new ItemSet(s);
280.         seq.getItemSetList().add(itemSet);
281.
282.         if (isLargerThanMinSupport(s, totalSeqs)) {
283.             tempSeqList = new ArrayList<>();
284.             for (Sequence s2 : totalSeqs) {
285.                 // 判断单一项是否包含于在序列中, 包含才进行提取操作
286.                 if (s2.strIsContained(s)) {

```

```

287.         tempSeq = s2.extractItem(s);
288.         tempSeqList.add(tempSeq);
289.     }
290. }
291.
292.         totalFrequentSeqs.add(seq);
293.         recursiveSearchSeqs(seq, tempSeqList);
294.     }
295. }
296.
297.     printTotalFreSeqs();
298. }
299.
300. /**
301.  * 按模式类别输出频繁序列模式
302.  */
303. private void printTotalFreSeqs() {
304.     System.out.println("序列模式挖掘结果: ");
305.
306.     ArrayList<Sequence> seqList;
307.     HashMap<String, ArrayList<Sequence>> seqMap = new HashMap<>();
308.     for (String s : singleItems) {
309.         seqList = new ArrayList<>();
310.         for (Sequence seq : totalFrequentSeqs) {
311.             if (seq.getItemSetList().get(0).getItems().get(0).equals(s)) {
312.                 seqList.add(seq);
313.             }
314.         }
315.         seqMap.put(s, seqList);
316.     }
317.
318.     int count = 0;
319.     for (String s : singleItems) {
320.         count = 0;
321.         System.out.println();
322.         System.out.println();
323.
324.         seqList = (ArrayList<Sequence>) seqMap.get(s);
325.         for (Sequence tempSeq : seqList) {
326.             count++;
327.             System.out.print("<");
328.             for (ItemSet itemSet : tempSeq.getItemSetList()) {
329.                 if (itemSet.getItems().size() > 1) {
330.                     System.out.print("(");
331.                 }
332.
333.                 for (String str : itemSet.getItems()) {
334.                     System.out.print(str + " ");
335.                 }
336.
337.                 if (itemSet.getItems().size() > 1) {
338.                     System.out.print(")");
339.                 }
340.             }
341.             System.out.print(">, ");
342.
343.             // 每5个序列换一行
344.             if (count == 5) {
345.                 count = 0;
346.                 System.out.println();
347.             }
348.         }
349.     }
350. }
351. }
352.
353. }

```

调用类Client.java:

```

[java]
01. package DataMining_PrefixSpan;
02.
03. /**
04.  * PrefixSpan序列模式挖掘算法
05.  * @author lyq
06.  *
07.  */
08. public class Client {

```

```
09.     public static void main(String[] args){
10.         String filePath = "C:\\Users\\lyq\\Desktop\\icon\\input.txt";
11.         //最小支持度阈值率
12.         double minSupportRate = 0.4;
13.
14.         PrefixSpanTool tool = new PrefixSpanTool(filePath, minSupportRate);
15.         tool.prefixSpanCalculate();
16.     }
17. }
```

输出的结果：

```
[java]
01. 原始序列数据:
02. <(b d )c b (a c )>
03. <(b f )(c e )b (f g )>
04. <(a h )(b f )a b f >
05. <(b e )(c e )d >
06. <a (b d )b c b (a d e )>
07. 序列模式挖掘结果:
08.
09.
10. <a >, <a a >, <a b >, <a b a >, <a b b >,
11.
12.
13. <b >, <b a >, <b b >, <b b a >, <b b c >,
14. <b b f >, <b c >, <b c a >, <b c b >, <b c b a >,
15. <b c d >, <b (c e )>, <b d >, <(b d )>, <(b d )a >,
16. <(b d )b >, <(b d )b a >, <(b d )b c >, <(b d )c >, <(b d )c a >,
17. <(b d )c b >, <(b d )c b a >, <b e >, <b f >, <(b f )>,
18. <(b f )b >, <(b f )b f >, <(b f )f >,
19.
20. <c >, <c a >, <c b >, <c b a >, <c d >,
21. <(c e )>,
22.
23. <d >, <d a >, <d b >, <d b a >, <d b c >,
24. <d c >, <d c a >, <d c b >, <d c b a >,
25.
26. <e >,
27.
28. <f >, <f b >, <f b f >, <f f >,
```

经过比对，与上述表格中的结果完全一致，从结果中可以看出他的递归顺序正是刚刚我所想要的那种。

算法实现时的难点

我在实现这个算法时确实碰到了不少的问题，下面一一列举。

- 1、Sequence序列在判断或者提取单项和组合项的时候，情况少考虑了，还有考虑到了处理的方式又可能错了。
- 2、递归的顺序在最早的时候考虑错了，后来对递归的顺序进行了调整。
- 3、在算法的调试时遇到了，当发现某一项出现问题时，不能够立即调试，因为里面陷入的递归层次实在太深，只能自己先手算此情况下的前缀，后缀序列，然后自己模拟出1个Seq调试，在纠正extract方法时用的比较多。

我对PrefixSpan算法的理解

实现了这个算法之后，再回味这个算法，还是很奇妙的，一个序列，通过从左往右的扫描，通过各个项集的子集，能够组合出许许多多的的序列模式，然后进行挖掘，PrefixSpan通过递归的形式全部找出，而且效率非常高，的确是个很强大的算法。

PrefixSpan算法整体的特点

首先一点，他不会产生候选序列，在产生投影数据库的时候(也就是产生后缀子序列),他的规模是不不断减小的。PrefixSpan采用分治法进行序列的挖掘，十分的高效。唯一比较会有影响的开销就是在构造后缀子序列的过程，专业上的名称叫做构造投影数据库的时候。


- [上一篇](#) GSP序列模式分析算法
- [下一篇](#) CBA算法---基于关联规则进行分类的算法

顶
4

踩
1




查看评论



shanpi6361

4楼 2017-03-21 00:02发表

请问这个程序怎么把多个字符识别为序列中的一个元素啊...



abcdefgh123321

3楼 2016-05-02 16:35发表

想问下 每个频繁序列的支持度 想计算, 怎么改代码啊



baidu_31887577

2楼 2016-04-16 23:46发表


您好, 请问这个程序是在哪个软件运行的?



Android路上的人

Re: 2016-04-17 20:22发表


回复baidu_31887577: 在Eclipse上跑就可以了, 简单的java程序



abcdefgh123321

1楼 2016-03-22 16:57发表


您好, 想请问下最大项集个数您没有设定么? 没有找到相关的参数。



Android路上的人

Re: 2016-03-22 19:44发表

回复abcdefgh123321: 好像没有设置吧,这个程序写的比较久了,你可以再仔细查查我的程序.



abcdefgh123321

Re: 2016-03-23 11:33发表

回复Android路上的人: 好的 谢谢楼主 我试着加加参数

发表评论

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

* 以上用户言论只代表其个人观点, 不代表CSDN网站的观点或立场

PrefixSpan算法详解




oppeuro

2016-12-12 21:24

1075

序列模式的概念最早是由Agrawal和Srikant 提出的。 动机：大型连锁超市的交易数据有一系列的用户事务数据库，每一条记录包括用户的ID，事务发生的时间和事务涉及的项目。如果能在其中挖掘涉及事务间关联关系的模式，即用户几次购买行为间的联系，可以采取更有针对性的营销措施。

序列模式PrefixSpan算法介绍



textboy

2016-09-13 15:35

2549

序列 序列(sequence)是一组排好序的项集，不一定是直接连续的，但依然满足次序。序列模式的元素还可以是一个项集，如一组页面序列。序列模式挖掘比关联挖掘能得到更深刻的知识。 序列模式 sequence pattern mining，针对Frequent Sequences，典型的应用...



从新人到AI工程师，我用了4个月

学习计划如下....

PrefixSpan算法的介绍

2013-09-23 15:07 614KB

下载



FreeSpan 和 PrefixSpan 算法学习



Together_CZ 2017-04-13 16:30 842

今天遇到两个比较有名的在频繁模式挖掘领域中经常会使用到的算法：FreeSpan 和 PrefixSpan 算法，说起来，我对这两个算法只能说是了解，但却不能说很明白，正好今天遇到了就索性，查一下资料，整理一下作为今天关于这两个算法的学习笔记，文章末尾会给出来查询使用的资料链接：&#...

序列模式挖掘



jiary5201314 2016-05-04 22:34 2532

所谓序列模式，我的定义是：在一组有序的数据列组成的数据集中，经常出现的那些序列组合构成的模式。跟我们所熟知的关联规则挖掘不一样，序列模式挖掘的对象以及结果都是有序的，即数据集中的每个序列的条目在时间或空间上是有序排列的，输出的结果也是有序的。举个简单的例子来说明，关联规则一个经典的应用是计算超市...

不再死记硬背，一个公式学懂英文



英语长难句解读，记住这个公式就够了！

序列模式挖掘（AprioriAll和AprioriSome算法）

0 前提 Apriori算法：Fast algorithms for mining asso WeeYang 2016-10-12 08:21 2124
ciation rules (1994) (见参考文献) 序列模式挖掘是

由频繁项挖掘发展而来。 1 序言 序列模式 (sequential pattern) 挖掘最早由Agrawal等人提出，针对带有交易时间属性的...

关联分析之序列模式



rongyongfeikai2 2014-10-26 18:23 6437

1.序列挖掘的概念 在

数据挖掘中的模式发现（六）挖掘序列模式

序列模式挖掘序列模式挖掘(sequence pattern mini u013007900 2017-02-05 16:32 2077
ng)是数据挖掘的内容之一，指挖掘相对时间或其他

模式出现频率高的模式，典型的应用还是限于离散型的序列。。其涉及在数据示例之间找到统计上相关的模式，其中数据值以序列被递送。通常假设这些值是离散的，因此与时间序列挖掘是密切相关的，但时间...

4种序列模式挖掘算法的比较分析



u011860731 2015-08-15 20:49 4587

http://fpcheng.blog.51cto.com/2549627/829527 算法简介 AprioriAll算法属于Apriori类算法，其基本思想为首先遍历序列数据库生成候选序列并利用Apriori性质进行剪枝得到频繁序列。每次遍历都是通过连接...

机器学习：序列模式挖掘算法

----- ztf312 2016-03-14 21:30 3708 -----

题目：下面有关序列模式挖掘算法的描述，错误的是？ ...

序列模式挖掘——GSP算法

序列模式挖掘的基本概念 项目全集I、项集X和事务集合T的概念和文章关联规则挖掘——Apriori算法 中定义的一致。一个序列(Sequence)是一个有序的项集列表，这个有序通常是指时间有序。

prefixspan算法

http://blog.sina.com.cn/s/blog_6e85bf420100o66q.html prefixspan算法韩家炜老师在2004年提出的序列模式算法，该算法和他在2000提出的FP_growth算法有很大的相似之处，都避免产生候选序列。奇怪的是为什...

程序猿不会英语怎么行？英语文档都看不懂！

软件工程出身的英语老师，教你用数学公式读懂天下英文→



prefixspan序列模式挖掘算法的源代码

2010-05-21 19:54 19KB 下载



8大经典数据挖掘算法

大概花了将近2个月的时间，自己把18大数据挖掘的经典算法进行了学习并且进行了代码实现，涉及到了决策分类，聚类，链接挖掘，关联挖掘，模式挖掘等等方面。也算是对数据挖掘领域的小小入门了吧。下面就做个小小的总结，后面都是我自己相应算法的博文链接，希望能够帮助大家学习。 1.C4.5算法。C4.5算法...

PrefixSpan序列模式挖掘算法

更多数据挖掘代码：https://github.com/linyiqun/DataMiningAlgorithm 介绍 与GSP一样，PrefixSpan算法也是序列模式分析算法的一种，不过与前者不同的是PrefixSpan算法不产生任何的侯选集，在这点上可以说已经比GSP好很多了。Pr...

序列挖掘算法比较

AprioriAll + GSP + FreeSpan + PrefixSpan 1.基本概念 AprioriAll算法属于Apriori类算法，其基本思想为首先遍历序列数据库生成候选序列并利用Apriori性质进行剪枝得到频繁序列。 GSP (generalized sequen t...

数据挖掘中的模式发现（七）GSP算法、SPADE算法、PrefixSpan算法

这前两个算法真是出人意料地好理解GSP算法GSP算法是AprioriAll算法的扩展算法，其算法的执行过程和AprioriAll类似。其核心思想是：在每一次扫描(pass)数据库时,利用上一次扫描时产生的大序列生成候选序列,并在扫描的同时计算它们的支持度(support),满足支持度的候选序列作为下...

u013007900 2017-02-06 11:09 2823

序列模式PrefixSpan算法介绍

序列 序列(sequence)是一组排好序的项集，不一定是直接连续的，但依然满足次序。序列模式的元素还可以是一个项集，如一组页面序列。序列模式挖掘比关联挖掘能得到更深刻的知识。 序列模式 sequence pat

ternmining，针对Frequent Sequences，典型的应用...

程序猿不会英语怎么行？英语文档都看不懂！

软件工程出身的英语老师，教你用数学公式读懂天下英文→



序列挖掘算法比较



shuke1991 2016-09-13 16:26 752

AprioriAll + GSP + FreeSpan + PrefixSpan 1.基本概念 AprioriAll算法属于Apriori类算法，其基本思想为首先遍历序列数据库生成候选序列并利用Apriori性质进行剪枝得到频繁序列。GSP (generalized sequen t...

PrefixSpan算法详解



oppeuro 2016-12-12 21:24 1075

序列模式的概念最早是由Agrawal和Srikant 提出的。动机：大型连锁超市的交易数据有一系列的用户事务数据库，每一条记录包括用户的ID，事务发生的时间和事务涉及的项目。如果能在其中挖掘涉及事务间关联关系的模式，即用户几次购买行为间的联系，可以采取更有针对性的营销措施。

公司简介 | 招贤纳士 | 广告服务 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-660-0108 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 |

江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2017, CSDN.NET, All Rights Reserved

