机器学习大作业



题目：基于FP-growth算法高效发现频繁项集

学号： 2014216085

姓名： 李博古

邮箱： [602566046@qq.com](mailto:602566046@qq.com)

目录

[1.算法介绍 2](#_Toc417496580)

[2.算法描述及验证 3](#_Toc417496581)

[2.1 FP树 3](#_Toc417496582)

[2.2 构建FP树 5](#_Toc417496583)

[2.2.1创建FP树的数据结构 5](#_Toc417496584)

[2.2.2构建FP树 6](#_Toc417496585)

[2.3 从一棵FP树中挖掘频繁项集 10](#_Toc417496586)

[2.3.1 抽取条件模式基 11](#_Toc417496587)

[2.3.2 创建条件FP树 12](#_Toc417496588)

[3.算法应用：在微博源中发现一些频繁共现词 15](#_Toc417496589)

[3.1 微博API使用说明 15](#_Toc417496590)

[3.2本程序所用微博接口说明 16](#_Toc417496591)

[3.2 从微博中爬取数据并应用于算法 16](#_Toc417496595)

[4.总结 19](#_Toc417496596)

[5.参考文献 20](#_Toc417496597)

# 1.算法介绍

该算法称作FP-growth，在介绍算法前需介绍下Apriori原理，即如果某个项集是频繁的，那么它的所有子集也是频繁的，这个原理直观上并没有什么帮助，但是如果反过来看就有用了，也就是说如果一个项集是非频繁集，那么它的所有超集也是非频繁的（如图1-1所示）。而基于Apriori原理的Apriori算法在产生频繁模式完全集前需要对数据库进行多次扫描，同时产生大量的候选频繁集，这就使Apriori算法时间和[空间复杂度](http://baike.baidu.com/subview/540497/540497.htm)较大。在挖掘长频繁模式的时候性能往往低下，故Jiawei Han提出了FP-growth算法。

非频繁

**图1-1 图中给出了所有可能的项集，其中非频繁项集用黑色表示。由于集合｛2，3｝是非频繁的，因此｛0，2，3｝、｛1，2，3｝、｛0，1，2｝也是非频繁的**

FP-growth算法只需要对数据库进行两次扫描，而Apriori算法对于每个潜在的频繁项集都会扫描数据集判定给定模式是否频繁，因此FP-growth算法的速度要比Apriori算法快。在小规模数据集上，这不是什么问题，但当处理更大规模数据集时，就会产生较大问题。FP-growth只会扫描数据集两次，它发现频繁项集的基本过程如下：

1. 构建FP树；
2. 从FP树中挖掘频繁项集。

# 2.算法描述及验证

## 2.1 FP树

FP-growth算法将数据存储在一种成为FP树的紧凑数据结构中。FP代表频繁模式（Frequent Pattern）。一棵FP树看上去与计算机科学中其他树结构类似，但是它通过链接来连接相似元素,被连起来的元素项可以看成一个链表。图2-1给出了FP树的一个例子。

Φ

z:5

x:1

r:1

x:3

s:1

y:3

s:2

r:1

t:2

t:1

r:1

**图2-1 一棵FP树，包含着连接相似节点的链接**

同搜索树不同的是,一个元素项可以在一棵树中出现多次。FP树会存储项集的出现频率，而每个项集会以路径的方式存储在树中。存在相似元素的集合会共享树的一部分。只有当集合之间完全不同时，树才会分叉。树节点上给出集合中的单个元素及其所在序列中的出现次数，路径会给出该序列的出现次数。表2-1给出了用于生成图2-1中所示FP树的数据。

**表2-1 用于生产图2-1FP树的事务数据样例**

事务ID 事务中的元素项

001 r,z,h,j,p

002 z,y,x,w,v,u,t,s

003 z

004 r,x,n,o,s

005 y,r,x,z,q,t,p

006 y,z,x,e,q,s,t,m

在图2-1中，元素z出现了5次，集合｛r,z｝出现了1次。于是可以得出结论：z一定是自己本身或者和其他符号一起出现了4次，再看下z的其他可能性。集合｛t，s，y，x，z｝出现了2次，集合｛t，r，y，x，z｝出现了1次。元素项z的右标是5，表现出现了5次，所以它一定单独出现过一次。通过观察表2-1就能得出刚才的结论是否正确。由于这里设定的最小支持度（数据集中包含该项集的记录所占比例）设为3，故q和e并没用出现在树中。

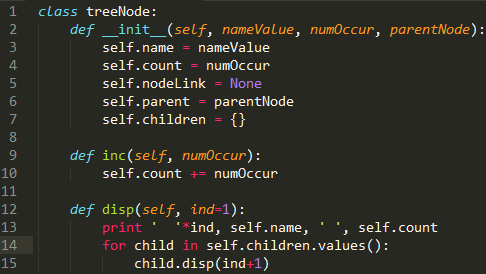
FP-growth算法的工作流程如下。首先构建FP树，然后利用它来挖掘频繁项集。为构建FP树,需要对原始数据集扫描两遍。第一遍对所有元素项的出现次数进行计数。运用Apriori 原理，即如果某元素是不频繁的，那么包含该元素的超集也是不频繁的，所以就不需要考虑这些超集。数据库的第一遍扫描用来统计出现的频率，而第二遍扫描中只考虑那些频繁元素。

**2.2 构建FP树**

在第二次扫描数据集时会构建一棵仲树。为构建一棵树，需要一个容器来保存树。

**2.2.1创建FP树的数据结构**

**程序清单2-1 FP树的类定义**



上面的程序给出了FP树中节点的类定义。类中包含用于存放节点名字的变量和1个计数值，nodeLink变量用于链接相似的元素项（参考图2-1中的箭头)。类中还使用了父变量parent来指向当前节点的父节点。指向父节点的指针用来根据给定叶子节点上溯整棵树。最后，类中还包含一个空字典变量，用于存放节点的子节点。程序清单2-1中包括两个方法，其中inc()对count变量增加给定值，而另一个方法disp()用于将树以文本形式显示。

现在FP树所需数据结构已经建好，下面就可以构建FP树了。

**2.2.2构建FP树**

除了图2-1给出的FP树之外，还需要一个头指针表来指向给定类型的第一个实例。利用头指针表，可以快速访问FP树中一个给定类型的所有元素。图2-2给出了一个头指针表的示意图。

这里使用一个字典作为数据结构，来保存头指针表。除了存放指针外，头指针表还可以用来保存FP树中每类元素的总数。

第一次遍历数据集会获得每个元素项的出现频率。接下来，去掉不满足最小支持度的元素项。再下一步构建FP树。在构建时，读入每个项集并将其添加到一条已经存在的路径中。如果该路径不存在，则创建一条新路径。每个事务就是一个无序集合。假设有集合｛z，x，y｝和｛y，z，r｝,那么在F P 树中 , 相同项会只表示一次。为了解决此问题，在将集合添加到树之前，需要对每个集合进行排序。排序基于元素项的绝对出现频率来进行。使用图2-2中的头指针节点值，对表2-1中数据进行过滤、重排序后的数据显示在表2-2中。

在对事务记录过滤和排序之后，就可以构建FP树了。从空集（符号为 Φ ) 开始，向其中不断添加频繁项集。过滤、排序后的事务依次添加到树中，如果树中巳存在现有元素，则增加现有元素的值；如果现有元素不存在，则向树添加一个分枝。对表2-2前两条事务进行添加的过程显示在图2-3中。

通过上面的叙述，已经大致了解了从事务数据集转换为FP树的基本思想，接下来通过代码来实现上述过程。

头指针表

z:5

r:3

x:4

y:3

s:3

t:3

Φ

z:5

x:1

r:1

x:3

s:1

y:3

s:2

r:1

t:2

t:1

r:1

**图2-2 带头指针表的FP树，头指针表作为一个起始指针来发现相似元素项**

**表2-2 将非频繁项移除并且排序后的事务数据集表**

事务ID 事务中的元素项 过滤及重排序后的事务

001 r,z,h,j,p z,r

002 z,y,x,w,v,u,t,s z,x,y,s,t

003 z z

004 r,x,n,o,s x,s,r

005 y,r,x,z,q,t,p z,x,y,r,t

006 y,z,x,e,q,s,t,m z,x,y,s,t

Φ

Φ

Φ

添加{z,r}

添加{z,x,y,s,t}

z:2

z:1

x:1

r:1

r:1

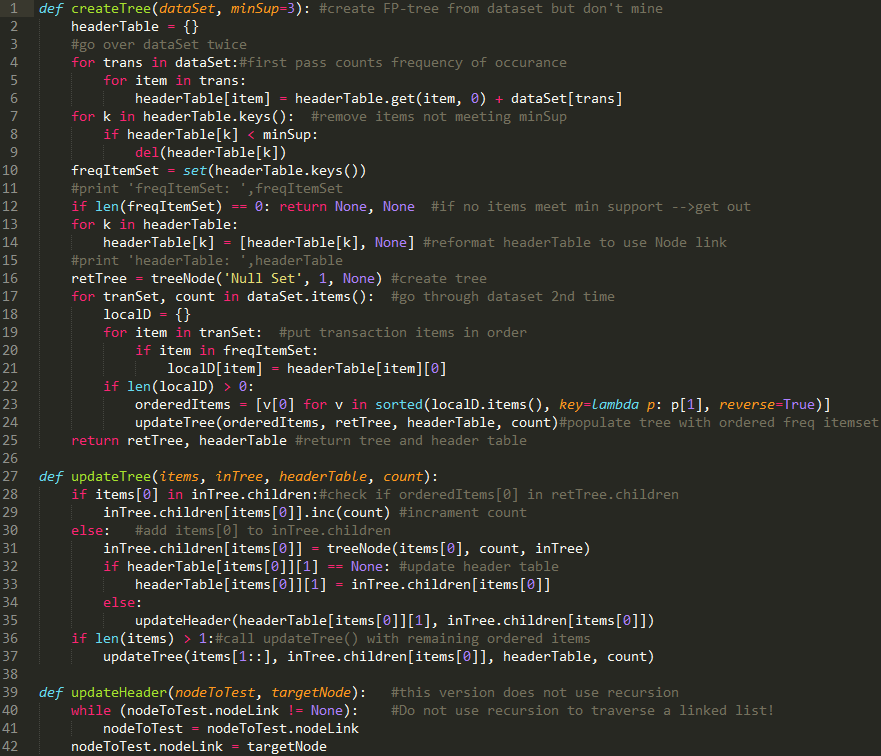
y:1

s:1

t:1

**图2-3 FP树构建过程的示意图，图中给出了使用表2-2中数据构建FP树的前两步**

**程序清单2-2 FP树构建函数**



上述代码中包含3个函数。第一个函数createTree() 使用数据集以及最小支持度作为参数来构建FP树。树构建过程中会遍历数据集两次。第一次遍历扫描数据集并统计毎个元素项出现的频度。这些信息被存储在头指针表中。接下来，扫描头指针表删掉那些出现次数少于minSup的项。如果所有项都不频繁，就不需要进行下一步处理。接下来，对头指针表稍加扩展以便可以保存计数值及指向每种类型第一个元素项的指针。然后创建只包含空集合 Φ的根节点。最后，再一次遍历数据集，这次只考虑那些频繁项。这些项已经如表2-2所示那样进行了排序,然后调用updateTree()方法。接下来讨论函数updateTree()。

为了让FP树生长,需调用updateTree(),其中的输人参数为一个项集。图2-3给出了updateTree()中的执行细节。该函数首先测试事务中的第一个元素项是否作为子节点存在。如果存在的话，则更新该元素项的计数；如果不存在，则创建一个新的treeNode并将其作为一个子节点添加到树中。这时，头指针表也要更新以指向新的节点。更新头指针表需要调用函数updateHeader (), 接下来会讨论该函数的细节。updateTree ( )完成的最后一件事是不断迭代调用自身， 每次调用时会去掉列表中第一个元素.。

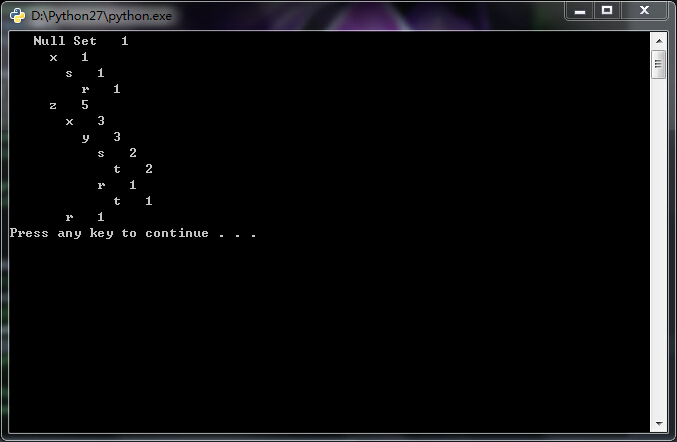
程序清单12-2中的最后一个函数是updateHeader ()，它确保节点链接指向树中该元素项的每一个实例。从头指针表的nodeLink开始，一直沿着nodeLink直到到达链表末尾。这就是一个链表。当处理树的时候，一种很自然的反应就是迭代完成每一件事。当以相同方式处理链表时可能会遇到一些问题，原因是如果链表很长可能会遇到迭代调用的次数限制。

在运行上例之前，还需要一个真正的数据集。loadSimpDat()函数会返回一个事务列表。这和表2-1中的事务相同。后面构建树时会使用createTree（）函数，而该函数的输人数据类型不是列表。其需要的是一部字典，其中项集为字典中的键，而频率为每个键对应的取值。createInitSet()用于实现上述从列表到字典的类型转换过程。

**程序清单2-3 简单数据集及数据包装**

![C:\Users\Administrator\AppData\Roaming\Tencent\Users\602566046\QQ\WinTemp\RichOle\P5%92_`%B5257]~T{B]2](R.png](data:image/png;base64,)

使用disp()方法给出树的文本表示结果:



上面给出的元素项及其对应的频率计数值,其中每个缩进表示所处的树的深度。其结果显示与图2-2所示的树等价。接下来使用它进行频繁项集挖掘。

**2.3 从一棵FP树中挖掘频繁项集**

有了FP树之后，就可以抽取频繁项集了。这里的思路与Apriori算法大致类似，首先从单元素项集合开始，然后在此基础上逐步构建更大的集合。当然这里将利用FP树来做实现上述过程，不再需要原始数据集了。

从FP树中抽取频繁项集的三个基本步骤如下：

(1)从FP树中获得条件模式基；

(2)利用条件模式基，构建一个条件FP树；

(3)迭代重复步骤(1)步骤( 2 ) ,直到树包含一个元素项为止。

**2.3.1 抽取条件模式基**

首先从已经保存在头指针表中的单个频繁元素项开始。对于每一个元素项，获得其对应的条件模式基。条件模式基是以所查找元素项为结尾的路径集合。每一条路径其实都是一条前辍路径。简而言之，一条前缀路径是介于所査找元素项与树根节点之间的所有内容。

回到图2-2，符号r的前缀路径是{x,s}、{z,x,y} 和{z}。每一条前缀路径都与一个计数值关联。该计数值等于起始元素项的计数值，该计数值给了每条路径上r的数目。表2-3列出了上例当中每一个频繁项的所有前缀路径。

**表2-3 每个频繁项的前缀路径**

频繁项 前缀路径

z {}5

r {x,s}1,{z,x,y}1,{z}1

x {z}3,{}1

y {z,x}3

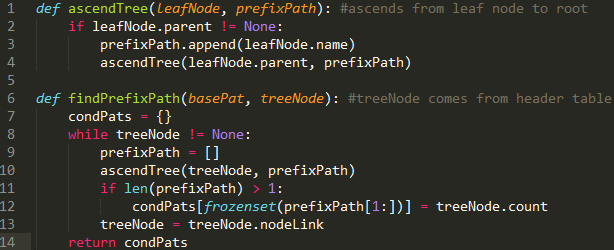
s {z,x,y}2,{x}1

t {z,x,y,s}2,{z,x,y,r}1

前缀路径将被用于构建条件FP树。为了获得这些前缀路径，可以利用先前创建的头.指针表来得到一种更有效的方法。头指针表包含相同类型元素链表的起始指针。一旦到达了每一个元素项，就可以上溯这棵树直到根节点为止。

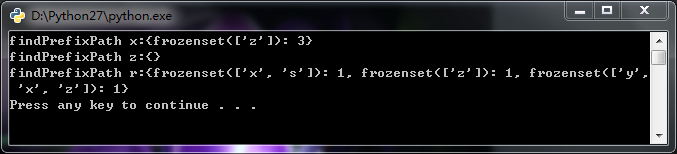
下面的程序清单给出了前缀路径发现的代码。

**程序清单2-4 发现已给定元素项结尾的所有路径的函数**



上述程序中的代码用于为给定元素项生成一个条件模式基，这通过访问树中所有包含给定元素项的节点来完成。当创建树的时候，使用头指针表来指向该类型的第一个元素项，该元素项也会链接到其后续元素项。函数findPrefixPath()遍历链表直到到达结尾。每遇到一个元素项都会调用ascendTree()来上溯FP树，并收集所有遇到的元素项的名称。该列表返回之后添加到条件模式基字典condPats中。

下面是使用之前构建的树的实际运行效果:



结果与表2-3中的结果一致。有了条件模式基之后，就可以创建条件FP树。

**2.3.2 创建条件FP树**

对于每一个频繁项，都要创建一棵条件FP树。我们会为z、x以及其他频繁项构建条件树。可以使用刚才发现的条件模式基作为输入数据，并通过相同的建树代码来构建这些树。然后，我们会递归地发现频繁项、发现条件模式基，以及发现另外的条件树。举个例子来说，假定为频繁项t创建一个条件FP树，然后对似｛t,y｝、{t,x},…重复该过程。元素项t的条件FP树的构建过程如图2-4所示。

**t的条件FP树**

条件模式基:{y,x,s,z}:2,{y,x,r,z}:1

最小支持度=3 去掉：s&r

Φ

Φ

Φ

加入{y,x,z}:1

加入{y，x，z}:2

y:3

y:2

x:3

x:2

z:3

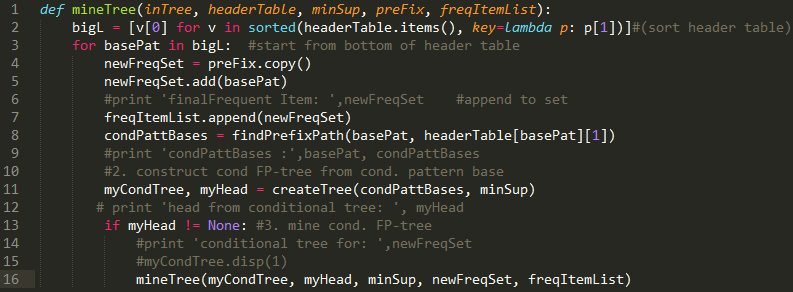
z:2

**图2-4 t的条件树的创建过程。最初以空集作为根节点。接下来,原始的集合｛y,x,s,z｝中的集合{y,x,z}被添加进来。因为不满足最小支持度要求，字符s并没有添加进来。类似的,{y,x,z}也从原始集合｛y,x,r,z｝中添加进来**

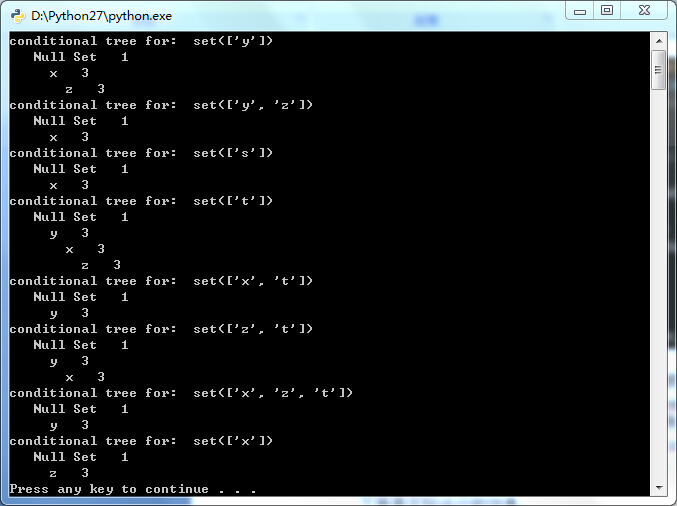
在图2-4中，注意到元素项s以及r是条件模式基的一部分，但是它们并不属于条件FP树。这是因为单独来看它们都是频繁项，但是在t的条件树中，它们却不是频繁的，也就是说，{t,r}及{t,s}是不频繁的。

接下来，对集合{t,z}、{t,x}以及{t,y}来挖掘对应的条件树。这会产生更复杂的频繁项集。该过程重复进行，直到条件树中没有元素为止。实现代码如下所示。

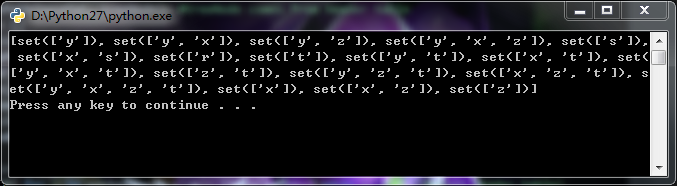
**程序清单2-5 递归查找频繁项集的mineTree函数**



程序首先对头指针表中的元素项按照其出现频率进行排序（从小到大)。然后，将每一个频繁项添加到频繁项集列表freqItemList中。接下来，递归调用程序清单2-4中的findPrefixPath(）函数来创建条件基。该条件基被当成一个新数据集输送给createTree()函数。这里为函数createTree()添加了足够的灵活性，以确保它可以被重用于构建条件树。最后，如果树中有元素项的话，递归调用mineTree()函数。

下面是实际运行的效果

对应的频繁项集：



可以看到，返回项集与条件FP树相匹配。到现在为止，完整的FP-growth算法已经可以运行，接下来在一个真实的例子上测试一下运行效果。将看到是否能从新浪微博中获得一些常用词。

# 3.算法应用：在微博源中发现一些频繁共现词

## 3.1 微博API使用说明

在使用微博API之前，需要申请微博的APPKEY，登录微博开发平台，创建一个应用后会生成对应的appkey,app\_secret,然后在根据这两个获取OAuth 2.0授权的代码（code）即可调用微博API。授权页面如下：



## 3.2本程序所用微博接口说明

# statuses/friends\_timeline

获取当前登录用户及其所关注用户的最新微博

## 支持格式

**JSON**

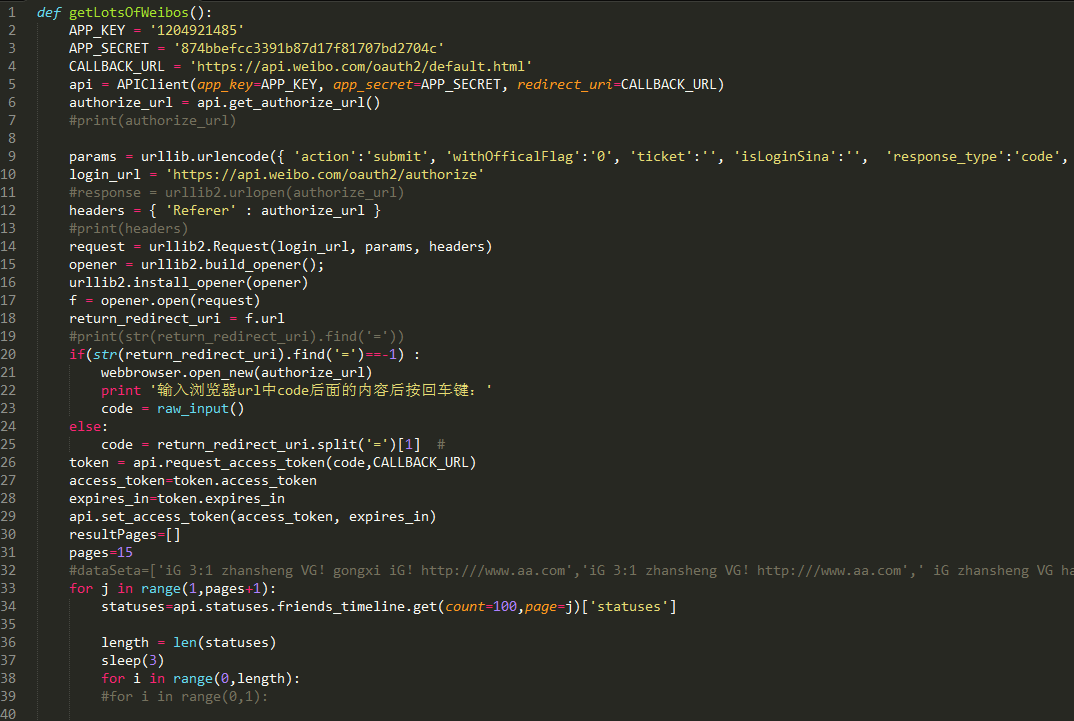
## 请求参数

|  |  |  |  |
| --- | --- | --- | --- |
|  | **必选** | **类型及范围** | **说明** |
| **source** | false | string | 采用OAuth授权方式不需要此参数，其他授权方式为必填参数，数值为应用的AppKey。 |
| **access\_token** | false | string | 采用OAuth授权方式为必填参数，其他授权方式不需要此参数，OAuth授权后获得。 |
| **since\_id** | false | int64 | 若指定此参数，则返回ID比since\_id大的微博（即比since\_id时间晚的微博），默认为0。 |
| **max\_id** | false | int64 | 若指定此参数，则返回ID小于或等于max\_id的微博，默认为0。 |
| **count** | false | int | 单页返回的记录条数，最大不超过100，默认为20。 |
| **page** | false | int | 返回结果的页码，默认为1。 |
| **base\_app** | false | int | 是否只获取当前应用的数据。0为否（所有数据），1为是（仅当前应用），默认为0。 |
| **feature** | false | int | 过滤类型ID，0：全部、1：原创、2：图片、3：视频、4：音乐，默认为0。 |
| **trim\_user** | false | int | 返回值中user字段开关，0：返回完整user字段、1：user字段仅返回user\_id，默认为0。 |

## 3.2 从微博中爬取数据并应用于算法

下面的代码即从OAuth 2.0获得授权后使用微博API爬取微博数据，然后将获得的微博进行分词，从中发现频繁出现的共现词的程序。

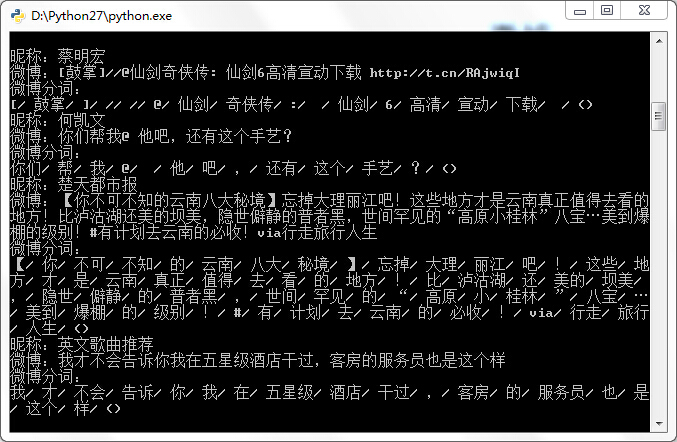
**程序清单3-1 访问微博API爬取数据并分析**



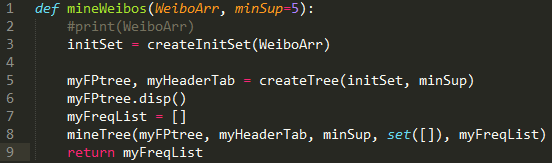


数据集总共包括1500条微博（共15页，每页100条微博）。

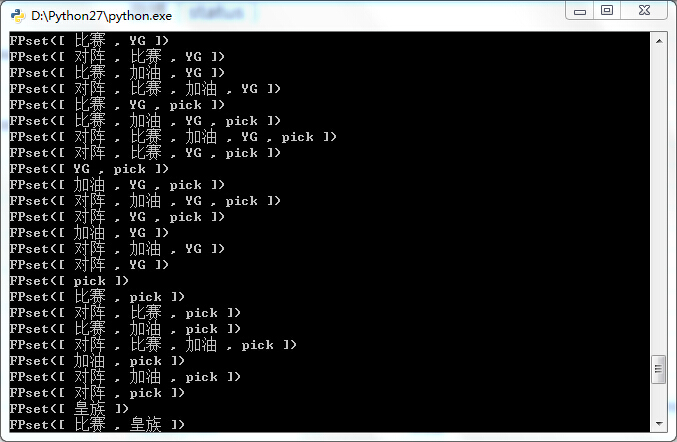
下面是抓取过程中的部分微博内容及分词结果：



下面只需将这些获取的数据集运行在FP-growth算法即可。代码如下：



调用listOfTerms = mineWeibos(getLotsOfWeibos(),10)构建FP树（10代表支持度）并对其进行挖掘；printResult(listOfTerms)打印结果如下：



由于本人收听的微博大多数与电子竞技有关，而当天正在举行英雄联盟游戏的比赛，而这些都可以从获得的结果频繁集中体现出来，并且从获取的结果中我们可以发现比赛对阵的双方各是哪两个队伍（皇族和YG）。这说明对于这1500条微博数据，在当前支持度（10）下的FP-growth算法得到了比较不错的结果。

# 4.总结

FP-growth算法是一种用于发现数据集中频繁模式的有效方法。FP-growth算法利用Apriori原则，执行更快。Apriori算法产生候选项集，然后扫描数据集来检查它们是否频繁。由于只对数据集扫描两次，因此FP-growth算法执行更快。在FP-growth算法中，数据集存储在一个称为FP树的结构中。FP树构建完成后，可以通过查找元素项的条件基及构建条件FP树来发现频繁项集。该过程不断以更多元素作为条件重复进行， 直到FP树只包含一个元素为止。

可以使用FP-growth算法在多种文本文档中查找频繁单词。新浪微博网站为开发者提供了大量的API来使用他们的服务。利用Python模块weibo可以很容易访问新浪微博数据。在微博源上对某个话题应用FP-growth算法，可以得到一些有关该话题的摘要信息。当然频繁项集生成还有其他的一些应用，比如购物交易、医学诊断及大气研究等。

通过本次的学习,我学习了Apriori算法及基于其原理的FP-growth算法,通过简单的数据集,实现并验证了算法过程,并在最后将其运用于挖掘新浪微博中频繁出现的共现词。在这个过程中我还学习并使用了微博API来获取真实数据，熟悉了Python的语法。

# 5.参考文献

[1]Peter Harrington, Machine Learning in Action

[2]Shui Wang,Le Wang,An Implementation of FP-Growth Algorithm Based

on High Level Data Structures of Weka-JUNG Framwork, Journal of Convergence Information Technology ,Volume 5, Number 9. November 2010

[3] H. Li, Y. Wang, D. Zhang, M. Zhang, E. Chang, “PFP: ParaUel FP-growth for Query Recommendation,” RecSys’08，Proceedings of the 2008 ACM Conference on Recommender Systems

[4] J.Han, J. Pei, Y. Yin，R. Mao， Mining Frequent Patterns without Candidate Generation: A Frequent-Pattem Tree Approach，” Data Mining and Knowledge Discovery .8 (2004)，53-87.