Data-Mining for Software Engineering: Frequent Itemsets on Source Code

Martin Monperrus

version of Nov 26, 2012

In one slide

- We can use data-mining techniques to document common patterns and detect bugs
- For instance, by mining itemsets of function/method calls
- And by using the Apriori algorithm to mine frequent itemsets

Related courses

- Mining Software Engineering Data, Queen's University, Canada
- Data Mining in Software Engineering, Universität Konstanz, Germany
- Mining Software Repositories, University of Victoria, Canada
- Topics in Mining Software Artifacts and Repositories, University of Alberta, Canada
- Mining Software Archives, Saarland University, Germany
- Software Evolution and Repository Mining, Hong Kong University of Science and Technology, China
- and others...

PR-Miner

postgresql-8.0.1/src/backend/catalog/dependency.c:
1733 getRelationDescription (StringInfo buffer, Oid relid)
1734 {
1735 HeapTuple relTup;
.....
1740 relTup = **SearchSysCache** (...);
.....
1796 **ReleaseSysCache** (relTup);
1797 }

A function-pair rule in PostgreSQL (extracted by PR-Miner). This rule appears 209 times in PostgreSQL code.

PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code, Li and Zhou, 2005

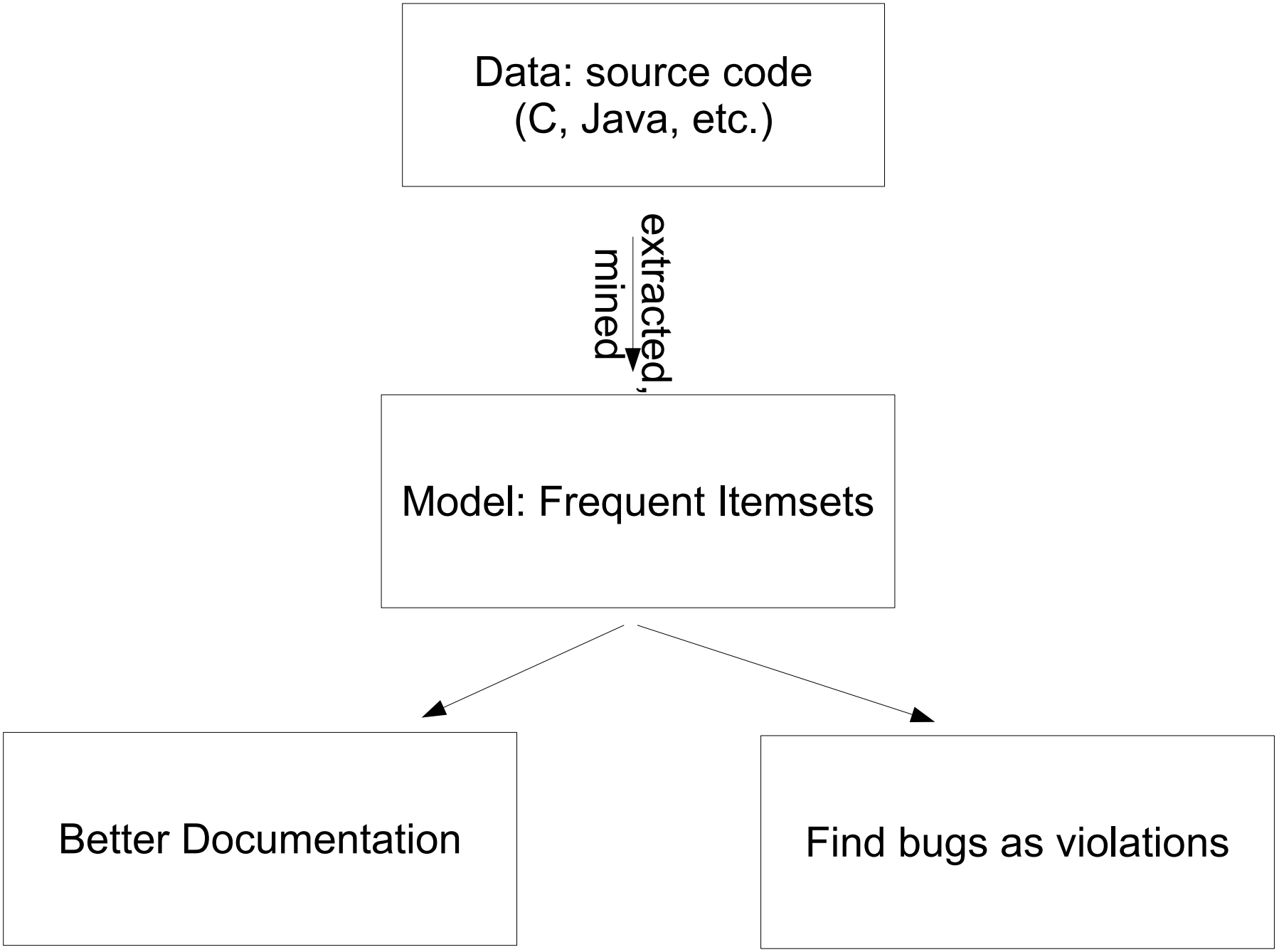
Another example

linux-2.6.11/drivers/scsi/3w-9xxx.c:

```
1964 int __devinit twa_probe(struct pci_dev *pdev, ...)
1965 {
1966 struct Scsi_Host *host = NULL;
.....
1985 host = scsi_host_alloc(...);
.....
2036 scsi_add_host(host, &pdev->dev);
.....
2069 scsi_scan_host(host);
.....
2088 }
```

A function-pair rule in Linux (extracted by PR-Miner). This rule appears 27 times.

The idea


Data: source code
(C, Java, etc.)
extracted,
mined
Model: Frequent Itemsets
Better Documentation
Find bugs as violations

Mapping function to itemsets

```
getRelationDescription (StringInfo buffer, Oid relid)
{
HeapTuple relTup;
......
relTup = SearchSysCache (...);
......
ReleaseSysCache (relTup);
}
```

-> {F-SearchSysCache, F-ReleaseSysCache}

Mapping function calls to itemsets (F-*)

```
getRelationDescription (StringInfo buffer, Oid relid)
{
HeapTuple relTup;
......
relTup = SearchSysCache (...);
......
ReleaseSysCache (relTup);
}
```

-> {F-SearchSysCache, F-ReleaseSysCache}

Mapping fields to itemsets (R-*)

```
void ll_stop(struct IsdnCardState *cs)
{
isdn_ctrl ic;
ic.command= ISDN_STAT_STOP;
ic.driver = cs->myid;
cs->iif.statcallb(&ic);
}
```

-> {R.isdn_ctrl.command, R.isdn_ctrl.driver, F-statcallb}

Mapping global variables to itemsets

```
void AlterTable()
{
   ...
   // lock is a global variable
   simple_heap_update (lock, ...);
    }
```

-> {G-lock, F-simple_heap_update}

Recap.

- One itemset per function
- Items depend on programming language, and desired granularity
  - F-*, R-*, G-*
- We obtain a (large) set of itemset:
  - {F-SearchSysCache, F-ReleaseSysCache}
  - {R.isdn_ctrl.command, R.isdn_ctrl.driver, F-statcallb}
  - ….
  - **{G-lock, F-simple_heap_update}**
  - **x 156324**

Frequent set mining

- A common way of inferring knowledge consists of mining "frequent sets"
- Frequent set mining is the identification of sets of items that often occur together
  - e.g. you buy bread and butter together in a supermarket
  - gene A is often found with gene B
  - you call method foo together with method bar
  - you override method X together with method Y

Terminology

Frequent Set Mining: Given a set of items I, a database of transactions D over I, and minimal support threshold t, find F(D, t).

- A **transaction** is a set of items and an identifier
- The **support** of a set X is the number of transactions that supports X
- A set is called **frequent** if its support is no less than a given absolute **mininal support threshold**.

The search space of all possible sets consists of 2^(|I|) sets. It is impossible to enumerate them.

tid	set of items
100	{beer, chips, wine}
200	{beer, chips}
300	{pizza, wine}
400	{chips, pizza}

Missing cover
Value of support?
Value of frequency?

Set	Cover	Support	Frequency
{}	{100, 200, 300, 400}	4	100%
{beer}	{100,200}	2	50%
{chips}	{100,200,400}	3	75%
{pizza}	{300,400}	???????	
{wine}	{100,300}	2	50%
{beer, chips}	{100,200}	2	50%
{beer, wine}	{100}	1	25%
{chips, pizza}	{400}	1	25%
{chips, wine}	{100}	1	25%
{pizza, wine}	{300}	1	25%
{beer, chips, wine}	{100}	???????	

Main property of item sets

If a set of items S is frequent i.e., appears in at least x % of the transactions, then every subset of S is also frequent (monotonicity or the a-priori trick)

$$X \in Y \Rightarrow support(Y) \leq support(X)$$

Apriori: intuition

1. find the frequent items sets of size 1

2. find the frequent pairs

3. find the the frequent triples

etc.

=> one pass on the data per level.

=> Max in memory: all frequent itemsets of size n

The apriori algorithm: overview

Loop:

- Calculate the frequency of all candidates (one pass over the data)
- Filter the non-frequent sets and display the frequent ones
- Generate candidates of size n+1 based on frequent sets of size n

```
Input configuration: 39 items, 1975 transactions, minsup = 0.27%

Candidates of size 1: 39
Frequent 1-itemsets: 5 / 39

Candidates of size 2: 10
Frequent 2-itemsets: 6 / 10

New candidates of size 3 : 4
Frequent 3-itemsets: 2 / 4

13 frequents sets for support 0.27%
```

Calculating the frequency

Frequency:

- For each candidate itemset
  - Count the number of transactions which contain it
- For each candidate itemset
  - Remove those whose cover is lower than the support

```
Input configuration: 39 items, 1975 transactions, minsup = 0.27%
New candidates: 4
...
[1, 5, 6] is contained in trans 1805 ( 6 1 5 7 11)
[1, 5, 6] is not contained in trans 1821 ( 11 19 1 5 18 3 0)
[1, 5, 6] is not contained in trans 1834 ( 5 6 1 2 0 4 3)
[1, 5, 6] is contained in trans 1834 ( 5 6 1 2 0 4 3)
[1, 5, 6] is not contained in trans 1894 ( 11 0 5 1)
[1, 5, 6] is not contained in trans 1896 ( 0 5 11 1)

++ Keep candidate: [1, 5, 6]  is: 0.31341772151898734
-- Remove candidate: [0, 5, 6]  is: 0.2612658227848101 < 27%
```

Candidate Generation:

- Create a new candidate itemset of size n+1 for each pair of frequent itemset os size n

```
Creating candidate [0, 5, 6]
   from [6, 5]
   from [6, 0]
Creating candidate [1, 5, 6]
   from [6, 5]
   from [1, 5]
Creating candidate [0, 1, 6]
   from [6, 0]
   from [1, 0]
Creating candidate [??????????]
   from [0, 5]
   from [1, 5]
New candidates: 4
++ Keep candidate: [0, 1, 5]  is: 0.2936708860759494
++ Keep candidate: [1, 5, 6]  is: 0.31341772151898734
Frequent 3-itemsets: 2
Creating candidate [0, 1, 5, 6]
   from [0, 1, 5]
   from [1, 5, 6]
-- Remove candidate: [0, 1, 5, 6]  is: 0.26126582278481014
13 frequents sets for support 0.27%
```

Frequent itemsets: Implementation

- Christian Borgelt's Apriori: http://www.borgelt.net/apriori.html
  - Fast
  - Stable and Mature
- Martin Monperrus's Java implementation: http://code.google.com/p/apriori-java/
- Patrick van Kouteren's Java implementation: http://www.vankouteren.eu/blog/2008/01/fr/

From frequent itemsets to documentation

- Analyze the itemset
- Compare with documentation

#2 java.lang.StringBuffer toString append <init> (224 | 2.4409%)

```
StringBuffer b = new StringBuffer(64);
b.append("name = " + name + " ");
b.append("type = " + type + " ");
return b.toString();
```

Similar to StringBuilder, result is obtained as a String.

Javadoc->Importance: **No** Bug if one misses: **Yes** Fills lack in API: **No**

#3 java.util.Iterator hasNext next (214 | 2.3319%)

```
Iterator iter = attributes.keySet().iterator();
while (iter.hasNext()) {
    list.add(iter.next());
}
```

Iterates a set using next() while it has items left.

Javadoc->Importance: **No** Bug if one misses: **No** Fills lack in API: **No**

#4 indexOf substring java.lang.String (130 | 1.4166%)

```
int pos = path.indexOf('?');
if (pos >= 0) {
    path = path.substring(0, pos);
}
```

Extracts a string to the first occurance of a given character.

Javadoc->Importance: **No** Bug if one misses: **No** Fills lack in API: **No**

The idea: define "violations", "bad smells" according to the frequent itemsets

PR-Miner proposal:

{a, b, d} , support: 3
{a}, support: 4

{a} => {b, d},
confidence: 3/4 = 75%
{a,b} => {d},
confidence 100%

Bug found by PR-Miner

{F-scsi_host_alloc, scsi_add_host} => {scsi_san_host}

linux-2.6.11/drivers/ieee1394/sbp2.c:

```
688 struct scsi_id_instance_data *sbp2_alloc_device
(struct unit_directory *ud)
689 {
......
692 struct scsi_id_instance_data *scsi_id = NULL;
......
745 scsi_host = scsi_host_alloc(...);
......
753 if (!scsi_add_host(scsi_host, &ud->device)) {
......
// scsi_scan_host(scsi_host) is missing!
764 }
```

Recap.

- It is powerful to use data-mining techniques for solving software engineering problems
- Technique: frequent itemsets (PR-Miner)