Frequent itemsets

A program that finds frequent itemsets and association rules based on baskets

Solution

The solution is split into subsections

Creating itemsets

Creating doubletons

Doubleton permutations were created by taking each singleton and combinging it with every remainging singleton in the list (creating a row for this singleton id in the structure mentioned later). This was an easy way to avoid duplicates and produced every possible doubleton.

Creating itemsets of any size

Creating larger itemsets was solved by taking the singletons and appending the larger (doubletons, tripletons etc) sets to each singleton. Two checks were added to avoid duplicates.

Firstly, singletons should not be appended by itemsets in which they are included. (17) and (17, 81) does not produce a valid tripleton.

Secondly, itemsets that are created must not already exists in a different form. (81, 17 , 202) should not be created if (17, 81, 202) already exists. This was solved by taking the smallest item id and looking through its row (as mentioned above) for duplicates. The list of singletons is ordered (ascending) and (17, 81, 202) will thus be created before (81, 17, 202), making it easy to find duplicates.

Counting itemsets

Itemsets of size 1

To find singletons the program simply iterates through every basket and increases a counter for each item id. This is done by letting the item id work as an index to an array of counters. When the rows are counted, itemsets are filtered out from the counters.

Itemsets of any size

Counting itemsets of an arbitrary size is trickier since there are more unique *items* to keep track of.
The solution used in this project is to look at each item id in a basket, find all itemsets that has this item id and check if they exist in this basket. This was done by creating an array (of itemset arrays) where the index is the item id that row starts with:

```
sets[17] = [(17, 55), (17, 99), (17, 505) ... ]
```

This datastructure contains few itemsets compared to the total itemsets being counted which made it easy finding itemsets and increasing their support if needed.

To increase throughput even further, threads were used and rows were calculated concurrently by different workers. Two small locks were used to achieve this. One on the job queue (bag of row indices) to synchronize work and one mutex lock (synchronization block) when updating the support of an itemset object:

```
# Increases the support for this item set
def increase_support
  @mutex.synchronize do
    @support += 1
  end
end
```

Finding association rules

Once all frequent itemsets are found, rules can be deducted.

This is done by creating all combinations of subsets from each itemset. The rule is then deducted as:

```
subset -> itemset\subset
```

Each rule is then validated by calculating its confidence by looking at the supports of the different itemsets in the rule. The solution uses the same datastructure as before to quickly find itemsets given item ids.

How to run

Prerequisites

The program is written in ruby, but uses jruby to fully utilize threading to increase speed. Jruby is highly recommended but ruby can of course be used.

Parameters

The only argument passable parameters are support and confidence:

- `support:` (optional) float (0, 1], defaults to 0.01
- `confidence:` (optional) float (0, 1] defaults to 0.5

note: confidence can only be used if support is given

Examples

Using jruby

note: when using jruby you are required to set the jvm heap size to an appropriate value depending on baskets

```
jruby -J-Xmx1024m lib/main.rb 0.01 0.5
```

Using ruby

note: if rvm (or rbenv etc) is installed, this will run as jruby because of .ruby-version

```
ruby lib/main.rb 0.02 0.75
```

The program will output all frequent itemsets with their support and assocation rules with their confidence.

Estimated run properties: 35 seconds runtime and 600MB RAM usage on 100k baskets with 1000 unique items.