Technichal Report of SF1R Recommender

May 23, 2011
iZENEsoft(Shanghai) Co., Ltd

Contents

1 SF1R Recommender API

Action: add_item
Description: Add a product item.

Request:

- * collection* (String): Add item in this collection.
- * resource* (Object): An item resource. Property key name is used as key. The corresponding value is the content of the property. Property ITEMID is required, which is a unique item identifier specified by client.

Reponse:

- header (Object): Property success gives the result, true or false.

Example:

```
{
  "resource" => {
    "ITEMID": "item_001",
    "name": "iphone",
    "link": "www.shop.com/product/item_001",
    "price": "5000"
    "category": "digital"
  }
}
```

Action: add_user
Description: Add a user profile.

Request:

- * collection* (String): Add user in this collection.
- * resource* (Object): A user resource. Property key name is used as key. The corresponding value is the content of the property. Property USERID is required, which is a unique user identifier specified by client..

Reponse:

- header (Object): Property success gives the result, true or false.

Example:

```
{
   "resource" => {
     "USERID": "user_001",
     "gender": "male",
     "age": "20",
     "area": "Shanghai"
   }
}
```

Action: do_recommend
Description: Get recommendation result.

Request:

- *collection* (String): Get recommendation result in this collection.
- *resource* (Object): A resource of the request for recommendation result.
    1. *rec_type_id* (Uint): recommendation type, now 6 types are supported, each with the id below:
        - 0 (Frequently Bought Together): get the items frequently bought together with input_items in one order
        - 1 (Bought Also Bought): get the items also bought by the users who have bought input_items
        - 2 (Viewed Also View): get the items also viewed by the users who have viewed input_items, in current version, it supports recommending items based on only one input item, that is, only input_items[0] is used as input , and the rest items in input_items are ignored
        - 3 (Based on Purchase History): get the recommendation items based on the purchase history of user USERID
        - 4 (Based on Browse History): get the recommendation items based on the browse history of user USERID
        - 5 (Based on Shopping Cart): get the recommendation items based on the shopping cart of user USERID, input_items is required as the items in shopping cart. If USERID is not specified for anonymous users, only input_items is used for recommendation.
    2. *max_count* (Uint = 10): max item number allowed in recommendation result.
    3. *USERID* (String): a unique user identifier.

4. *input_items* (Array): the input items for recommendation.
   - *ITEMID* (String): a unique item identifier.
5. *include_items* (Array): the items must be included in recommendation result.
   - *ITEMID* (String): a unique item identifier.
6. *exclude_items* (Array): the items must be excluded in recommendation result.
   - *ITEMID* (String): a unique item identifier.

Reponse:

- header (Object): Property success gives the result, true or false.
- resources (Array): each is an item in recommendation result.
  1. ITEMID (String): a unique item identifier.
  2. weight (Double): the recommendation weight, if this value is available, the items would be sorted by this value decreasingly.
  3. each item properties in add_item() would also be included here. Property key name is used as key. The corresponding value is the content of the property.

Example:

```
{
  "resource" => {
    "rec_type_id": "3",
    "max_count": "20",
    "USERID": "user_001"
  }
}


{
  "header": {"success": true},
  "resources" => [
    {"ITEMID": "item_001", "weight": 0.9, "name": "iphone"},
    {"ITEMID": "item_002", "weight": 0.8, "name": "ipad"},
    {"ITEMID": "item_003", "weight": 0.7, "name": "imac"}
    ...
  ]
}
```

Action: visit_item

Description: Add a visit item event.

Request:

- * collection* (String): Add visit item event in this collection.
- * resource* (Object): A resource for a visit event, that is, user USERID visited item ITEMID.
  1. *USERID* (String): a unique user identifier.
  2. *ITEMID* (String): a unique item identifier.

Reponse:

- header (Object): Property success gives the result, true or false.

Example:

```
{
  "resource" => {
    "USERID": "user_001",
    "ITEMID": "item_001"
  }
}
```

Action: update_item
Description: Update a product item.

Request:

- * collection* (String): Update item in this collection.
- * resource* (Object): An item resource. Property key name is used as key. The corresponding value is the content of the property. Property ITEMID is required, which is a unique item identifier specified by client.

Reponse:

- header (Object): Property success gives the result, true or false.

Example:

```
{
 "resource" => {
   "ITEMID": "item_001",
   "name": "iphone",
   "link": "www.shop.com/product/item_001",
   "price": "5000"
   "category": "digital"
 }
}
```

Action: remove_item
Description: Remove a product by item id.

Request:

- * collection* (String): Remove item in this collection.
- * resource* (Object): Only field ITEMID is used to remove the item.

Reponse:

- header (Object): Property success gives the result, true or false.

Example:

```
{
  "resource" => {
    "ITEMID": "item_001"
  }
}
```

Action: purchase_item
Description: Add a purchase item event.

Request:

- *collection* (String): Add purchase item event in this collection.
- *resource* (Object): A resource for a purchase event, that is, user USERID purchased items.
  1. * USERID* (String): a unique user identifier.
  2. * items* (Array): each is an item purchased.
     - *ITEMID* (String): a unique item identifier.
     - *price* (Double): the price of each item.
     - *quantity* (Uint): the number of items purchased.
  3. *order_id* (String): the order id.

Reponse:

- header (Object): Property success gives the result, true or false.

Example:

```
{
  "resource" => {
    "USERID": "user_001",
    "items": [
      {"ITEMID": "item_001", "price": 100, "quantity": 1},
      {"ITEMID": "item_002", "price": 200, "quantity": 2},
      {"ITEMID": "item_003", "price": 300, "quantity": 3}
    ],
    "order_id": "order_001"
  }
}
```

Action: get_user
Description: Get user from SF1 by user id.

Request:

- * collection* (String): Get user in this collection.
- * resource* (Object): Only field USERID is used to get the user.

Reponse:

- header (Object): Property success gives the result, true or false.
- resource (Object): A user resource. Property key name is used as key. The corresponding value is the content of the property.

Example:

```
{
  "resource" => {
    "USERID": "user_001"
  }
}


{
  "header": {"success": true},
  "resource" => {
    "USERID": "user_001",
    "gender": "male",
    "age": "20",
    "area": "Shanghai"
  }
}
```

Action: get_item
Description: Get item from SF1 by item id.

Request:

- * collection* (String): Get item in this collection.
- * resource* (Object): Only field ITEMID is used to get the item.

Reponse:

- header (Object): Property success gives the result, true or false.
- resource (Object): An item resource. Property key name is used as key. The corresponding value is the content of the property.

Example:

```
{
  "resource" => {
    "ITEMID": "item_001"
  }
}


{
  "header": {"success": true},
  "resource" => {
    "ITEMID": "item_001",
    "name": "iphone",
    "link": "www.shop.com/product/item_001",
    "price": "5000"
    "category": "digital"
  }
}
```

Action: remove_user
Description: Remove user by user id.

Request:

- * collection* (String): Remove user in this collection.
- * resource* (Object): Only field USERID is used to remove the user.

Reponse:

- header (Object): Property success gives the result, true or false.

Example:

```
{
  "resource" => {
    "USERID": "user_001"
  }
}
```

2 Architecture

As depicted in figure 1, `SF1R Recommender` is easily embedded into the `OSGI Bundle` based structure:

1. `Recommend Bundle` is used to serve the open API layer as services.
2. `Recommend Manager` is used to manage recommender-relevant data and processing flow.
3. `Recommend Algorithms` are all put to `iDMLib` to be called by `Recommend Manager`.

2.1 Recommend Manager

RecommendManager provides abstraction for both User and Item, while UserManager and ItemManager takes charge of storing user profiles and item profiles correspondingly. The management of visitation history and purchase history are charged by VisitManager and PurchaseManager. Currently, all the above four mentioned managers adopt `Tokyo Cabinet` as the storage engine while VisitManager will switch

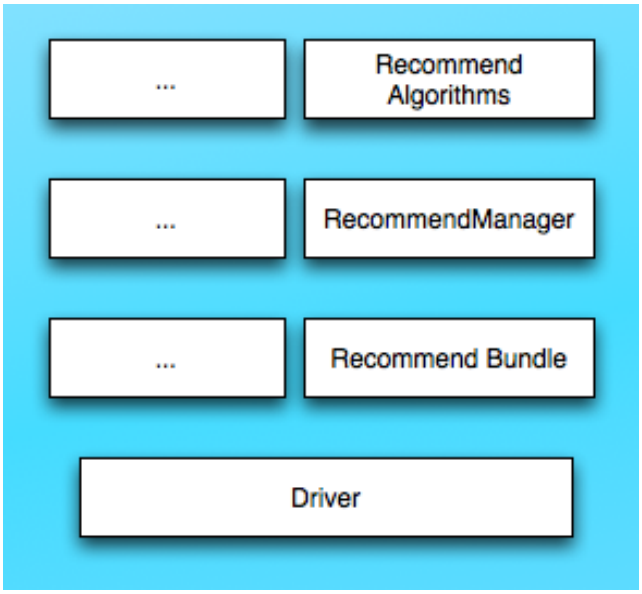

Figure 1: SF1R Recommender Architecture

the storage engine to something else because user click-through behavior requires an extremely frequently updating operation, and `Tokyo Cabinet` will perform extremely bad after 50M records' insertion—it is the short comings of all `BTree` bases storage engine. And since there are lots of instances of `Tokyo Cabinet` in `SF1R`, the updating bottleneck should be much smaller than the above mentioned limit—50M.

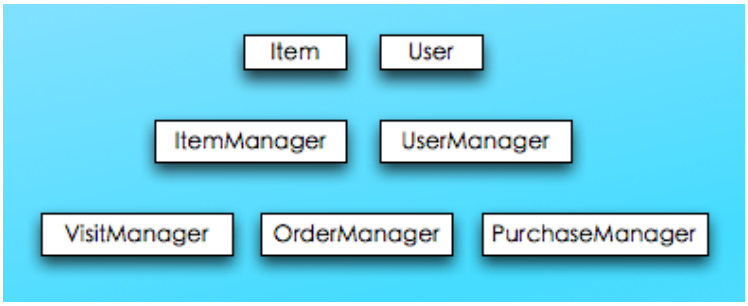

Figure 2: Recommend Manager

It should be noted that the identifiers for both User and Item are all string based ones, while they are required to be changed to integers within `SF1R` because of the memory consumption during recommendation computation tasks. An incremental policy for identifiers' allocation is reasonable because it could bring benefits for accessing elements very fast directly from an array-based data structure. Additionally, profiles for both user and item are required to be accessed, the identifiers mapping between external UserID, ItemID and internal ids used by recommendation algorithms should be persisted as well. Existing `IDManager` could serve as such usage while `Tokyo Cabinet` are used for these two kinds of persistence. Another issue

on identifiers is about the OrderID—Since there does not exist the requirements to access profiles for Orders, an incremental identifiers policy without persistence is used to serve that usage.

2.1.1 Importing Histories

Importing history data is necessary to resolve part of the so-called `cold start` problem. Similar as the existing indexing mechanism, `SF1R Recommender` supports three new kinds of SCD files to import history purchase data:

- UserSCD— `USERID` is a must in SCD files.

```
1 <USERID>user_001
2 <gender>male
3 <age>23
4 <area>Beijing
```

- ItemSCD— `ITEMID` is a must in SCD files.

```
1 <ITEMID>item_001
2 <name>iphone
3 <price>5000
4 <link>www.shop.com/product/item_001
5 <category>digital
```

- OrderSCD— `USERID` is a must in SCD files.

```
1 <USERID>user_001
2 <ITEMID>item_001
3 <DATE>2011-04-01
4 <order_id>order_001
5 <price>3000
6 <quantity>1
```

For other kinds of history data, such as user visitation, clickthrough, user wish lists,...,etc, since most of the available B2C providers do not have such mechanism to record them, `SF1R Recommender` does not support importing those kinds of data.

2.2 Storage Engine

As has been mentioned above, such storage engine as `Tokyo Cabinet` suffers from the updating problems because `BTree` data structure is not optimized for writing. Such problem is more serious within recommendation tasks' computation because before a distributed platform such as `Hadoop` is introduced, we only have single node solution that has to support up to 1M products. This will cause an extremely huge User-Item matrix and extremely frequently updating operation, so we could not use `Tokyo Cabinet` any more within the recommendation tasks—A `Log-Structured Merge Tree` based solution [1] is used to serve matrices within recommendation algorithms. Several essentials:

- Keys are stored in memory for fast lookups. All keys must fit in RAM.

- Writes are append-only, which means writes are strictly sequential and do not require seeking. Writes are write-through. Every time a value is updated the data file on disk is appended and the in-memory key index is updated with the file pointer.
- Read queries are satisfied with O(1) random disk seeks. Latency is very predictable if all keys fit in memory because there's no random seeking around through a file.
- For reads, the file system cache in the kernel is used instead of writing a complicated caching scheme.
- Old values are compacted or "merged" to free up space. Bitcask has windowed merges: Bitcask performs periodic merges over all non-active files to compact the space being occupied by old versions of stored data.
- The key to value index exists in memory and in the filesystem in hint files. The hint file is generated when data files are merged. On restart the index only needs to be rebuilt for non-merged files which should be a small percentage of the data.

3 Recommender Algorithms

As mentioned in 1, there are 6 kinds of recommendations:

- Frequently Bought Together
- Bought Also Bought
- Viewed Also View
- Based on Purchase History
- Based on Browse History
- Based on Shopping Cart

Except for the first category, the other recommendations could all be got through collaborative filtering, however, due to the real-time essentials of Viewed Also View, another kind of algorithm is introduced—CoVisitation.

3.1 Frequent Itemset

Frequently bought together is a typical kind of frequent itemset mining problem. Association Rule families are used to resolve such problem, typical algorithms include: `Apriori`, `FPTree` and their improvements. However, due to the fact that current architecture only support single node, and memory consumption should be controlled restrictedly, some other kinds of frequent itemset mining solutions are required to be prepared—An integration for finding extremal sets together with building inverted

index is eventually used because it is much faster than general `Apriori` and `FPTree` with a much lower memory consumption without sacrificing precision.

Finding extremal sets are state-of-the-art solution by Google in [2]. An itemset S1 is extremal among a collection of itemsets if there is no other itemset S2 in the collection such that $S1 \subset S2$, the algorithm is described in figure 3.

Algorithm 1 GetMaximalItemsetsCard(Dataset D): Find all maximal sets from a dataset D by using the cardinality constraint.

Require: D ordered by increasing itemset cardinality

```
1:  B ← ∅; c ← 0; O ← ∅
2:  for all i ∈ {1, ..., n} do
3:    for all j ∈ {1, ..., |D[i]|} do
4:      for all S ∈ O[D[i][j]] not marked as subsumed
        do
5:        if |S| > |D[i]| − j + 1 then
6:          break
7:        if D[i] properly subsumes S then
8:          mark S as subsumed
9:    if |D[i]| > c then
10:     Add each itemset S ∈ B to the end of O[S[1]]
11:     B ← ∅; c ← |D[i]|
12:   B ← B ∪ {D[i]}
13: return all elements of D not marked as subsumed
```

Figure 3: Finding Extremal Sets

According to experiments, such finding extremal sets algorithm could be finished within tens of seconds given 100K transactions, while `Apriori` or `FPTree` will require hours with much higher memory occupation. However, the extremal itemsets do not mean the eventual frequent itemsets. A post processing is required to be attached based on the results of finding extremal sets:

- Given each item from the extremal sets, find its subsets.
- Given each subset, find the co-occurance of all its items, and if it's above some threshold, store the itemset accordingly.

How to find the co-occurance quickly will decide the eventual performance of the frequent itemset finding process. Indicated from the idea of Bit-Slice Bloom Filtered Signature File in [3], we build an inverted index between ItemId and OrderId where term within IndexManager refers to ItemId and document identifiers refers to OrderIds, and the process of finding the co-occurance of several items from all orders could be simulated as an AND posting traverse procedure. Through such kinds of inverted index based solution, even if there are millions of order transactions, we could easily find a co-occurance within mili-seconds.

3.2 Covisitation

The idea of Co-visitqtion is very simple: maintaining an item-item co-visitation matrix 4, and whenever a click-through comes, update the value of corresponding row and column. As a result, the co-visitation matrix is an extremely frequently updated matrix. Bitcask style storage engine is inevitable. The recommendation results of Viewed Also View is just to retrieve a certain row according to item id.

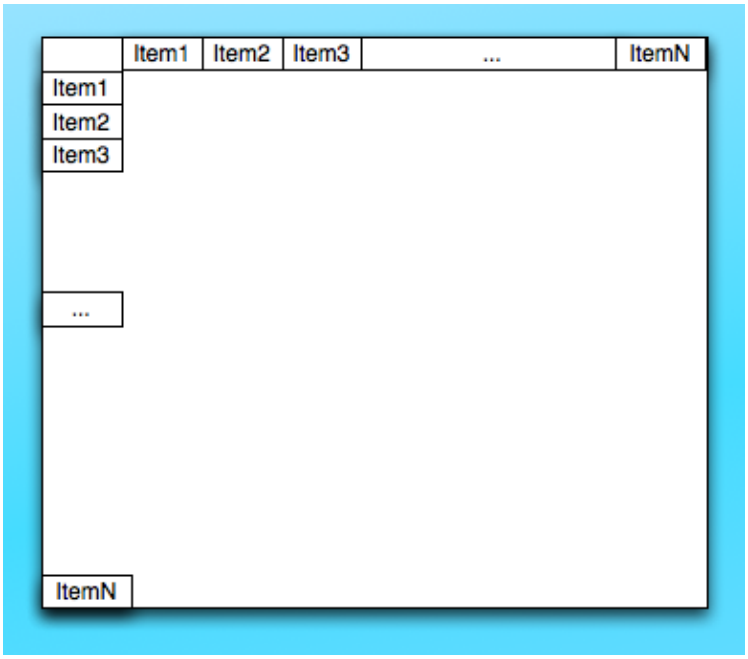

Figure 4: Co-Visitation Matrix

3.3 Collaborative Filtering

Collaborative filtering has been used in most commercial recommender systems. In the e-commerce area, all the user ratings are implicit ones because users never rate the products explicitly. Whenever a purchase happens, corresponding rating will be set to 1, while others are still kept to 0. As a result, 0-1 matrix is absolutely the most important case in e-commerce area. From the academic viewpoint, such problem is called as One Class Collaborative Filtering [5], or One Class Matrix Completion. The major challenge is how to deal with huge amount of missing value. In the current version of `SF1R Recommender`, an item-based collaborative filtering is used instead of a matrix factorization based One Class Matrix Completion, because it can always perform well on 0-1 matrix. Based on the discussion and experiments from Netflix Prize winners, their solutions perform even much worse than the simplest item-based collaborative filtering on 0-1 matrix. Another benefit of item-based collaborative filtering is, it is the only incremental solution that could be got from all collaborative filtering algorithms families—the solution of Incremental CF could be seen in [4].

One of the most important incremental solution is: we could provide a much better real-time recommendation result that has considered the latest purchase behaviors, while traditional collaborative filtering have to be built in a batch way, and during the intervals of batch building, the recommendation results are kept unchanged. Making traditional collaborative filtering approach be incremental is impossible: in the similarity matrix shown in figure 5, whenever a new session comes, we need a rebuilding for such matrix which means $O(m^2n)$ for m items and n users.

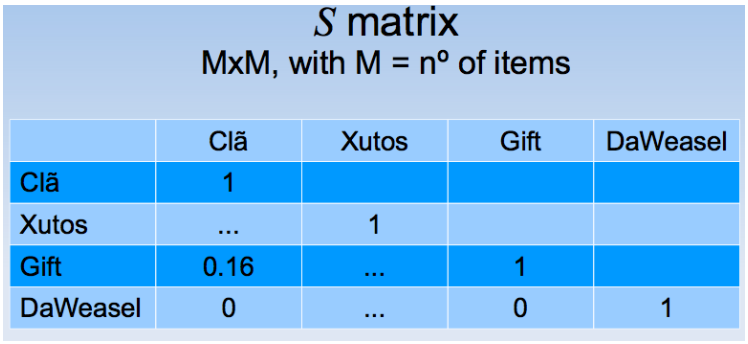

Figure 5: Similarity Matrix

Within incremental collaborative filtering, figure 6 shows how the similarity matrix is maintained. In the incremental collaborative filtering approach, another matrix $I \cap$

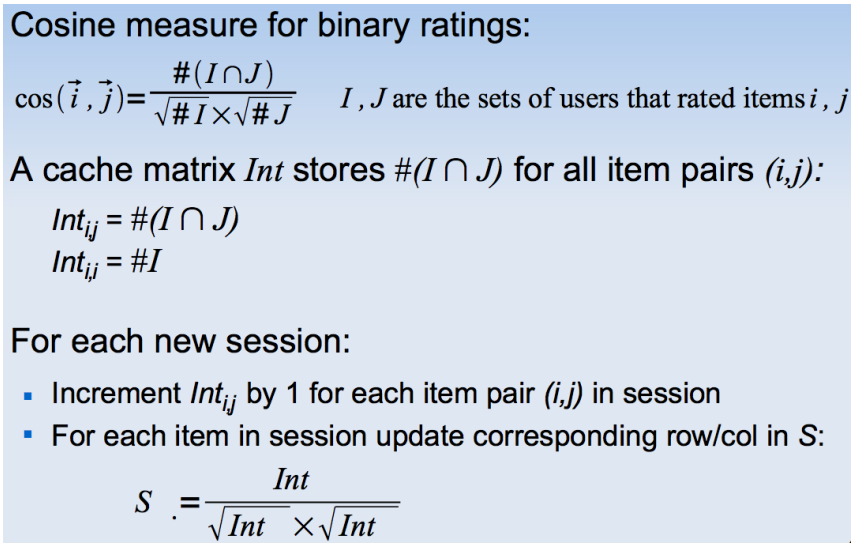

Figure 6: Incrementally Updating Similarity Matrix

J is also maintained, which is very similar with the Co-Visitation matrix mentioned previously. The only difference is: the Co-visitation matrix is updated whenever a click-through behavior happen, while $I \cap J$ is in fact a Co-Bought matrix which is updated whenever a purchase happens. From the implementation point of view, both of them share same data structure.

3.4 Future

The collaborative filtering approach suffers from several key problems:

- Cold start
- Diversity
- Novelty
- Serendipity

These problems would be gradually resolved through context-aware recommender system, which will be provided after basic collaborative filtering solution is applied to practical environments.

References

[1] http://downloads.basho.com/papers/bitcask-intro.pdf.

[2] R.J. Bayardo and B. Panda. Fast algorithms for finding extremal sets. 2011.

[3] B. Lan, B.C. Ooi, and K.L. Tan. Efficient indexing structures for mining frequent patterns. In Data Engineering, 2002. Proceedings. 18th International Conference on, pages 453–462. IEEE, 2002.

[4] C. Miranda and A.M. Jorge. Incremental collaborative filtering for binary ratings. In Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT'08. IEEE/WIC/ACM International Conference on, volume 1, pages 389–392. IEEE, 2008.

[5] R. Pan, Y. Zhou, B. Cao, N.N. Liu, R. Lukose, M. Scholz, and Q. Yang. One-class collaborative filtering. In Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on, pages 502–511. IEEE, 2008.