Technichal Report of SF1R Recommender

May 27, 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`.

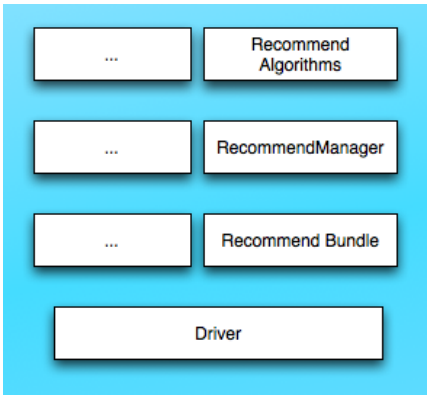

Figure 1: SF1R Recommender Architecture

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 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.

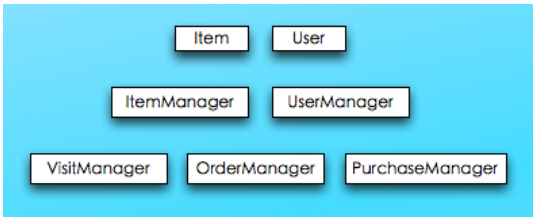

Figure 2: Recommend Manager

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.

```
<USERID>user_001
<gender>male
<age>23
<area>Beijing
```

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

```
<ITEMID>item_001
<name>iphone
<price>5000
<link>www.shop.com/product/item_001
<category>digital
```

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

```
<USERID>user_001
<ITEMID>item_001
<DATE>2011-04-01
<order_id>order_001
<price>3000
<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 [5], 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.

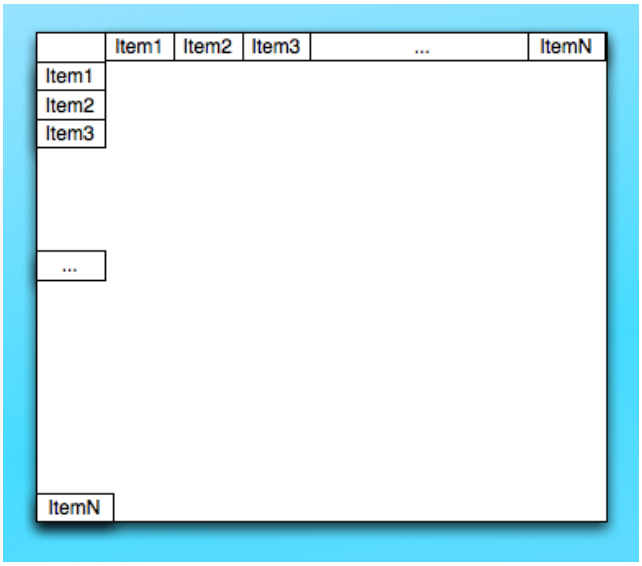

Figure 4: Co-Visitation Matrix

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 [11], 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 [10].

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.

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

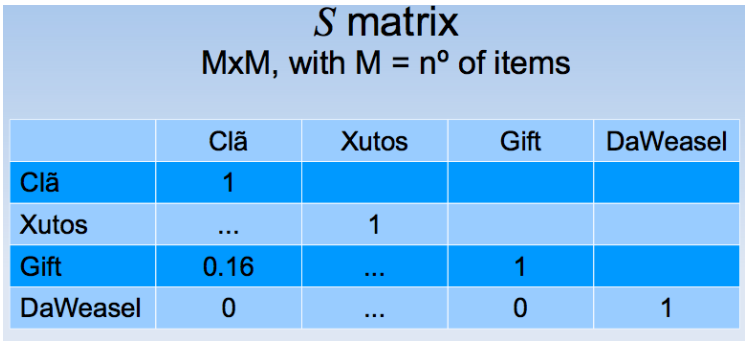

Figure 5: Similarity Matrix

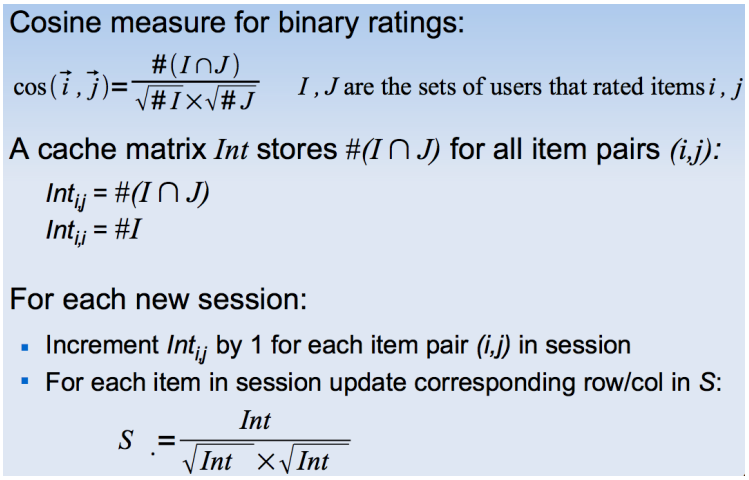

Figure 6: Incrementally Updating Similarity Matrix

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.

4 Evolution—Context-Aware Recommender

The collaborative filtering approach suffers from several key problems:

- Cold start
- Diversity
- Novelty
- Serendipity
- Concept drift

Traditionally recommender solutions still dive into collaborative filtering algorithms to meet those challenges, most of them are over-complicated without remarkable improvements. According to our insights, there are two reasons for why those works are trivial:

- All of the collabotative filtering algorithms, together with all of the content based recommender algorithms, are based on the assumption that users tend to like items globally similar to their histories, which means the future interest is predicted totally by past behaviors. Such assumption might not be true in practical situation.
- Collaborative filtering ignores the items' information, so it can not mine the items themselves. Although content based recommender is used to resolve this issue, they face challenges on missing data.

Based on our experiences of information retrieval, we found that the recommender problem is in fact a ranking problem and should have borrowed lots of results from information retrieval area—we should utilize those results to mine items themselves, so that the recommendation process is a search process without any cold start and sparsity problem. Additionally, our experiences on information discovery would help us provide items that suprise users. The discovery process should be based on the semantic similarity. As a result, we present a new recommender that contains these aspects:

- It is context-aware, which means it will provide recommendations based on user's current intention instead of past behaviors. In terminology, it's `context-aware`.
- It is based on semantic similarity engine and search engine, so that the problem of new items, sparsity, cold start does not exist. Additionaly, the serendipity and novelty will benefit from the so-called `Semantic`.
- It is real-time and adaptive during learning process when contextual information is not enough.

4.1 Context Aware Based on Semantic Similarity

To the best of our knowledge, there are no content-based recommender systems able to exploit the semantic text representations for learning profiles. The positive results obtained exploiting the advanced text representations in several tasks, such as semantic relatedness, text categorization and retrieval, suggest that similar positive results could be also obtained in the recommendation task. It seems a promising research area. In [12], an `ESA`-like(Explicit Semantic Analysis which is under development for document similarity) content based recommender is constructed, which could be looked on as an experiment on integrating semantic text analysis with recommender algorithms. Given our own insights on both semantic analysis and recommendation algorithms, we present a context-aware recommender based on semantic similarity which is composed of these steps:

- When a user comes to a web site, they immediately begin to establish their current context through

  1. Entering a query into a search box
  2. Navigating to a product
  3. Landing the site through external search engine such as Google or Baidu

  All of the above information is stored as the user's current context vector, which is a hybrid vector of terms and items with weights on each entry reflecting how strongly that entry reflects the user's current context. As a user enters queries or click navigation links, the vector entries corresponding to the terms and items are incremented to capture user's intention. As these actions move further into the past, the corresponding entries are decremented or decayed.

- Refine the context vector into an intent vector based on semantic similarities got through similarity engine. Eg: in the context vector, the entries contain `Nikon D60` camera, to create the intent vector, the system looks at semantic similarities between terms and items from context vecto to other terms, say, `high-resolution` may be highly associated with the term `Nikon`, then the intent vector is thus incremented at the entry corresponding to `high-resolution`. The ability to translate context into intent is based on semantic similarity engine.

- After intent vector has been got, there are two approaches to get recommendations:

  1. Identify the cluster of users who share affinity to the current intent, as well as those users who exhibit behavior most like the current user within the context of that intent. The user cluster is represented by a user vector and each user entry in the vector may have a weight indicating how strong of a cluster he is to the current user in this context. The users are clustered periodically using sematic similarity join algorithms, which will be described later. Recommendations are those with highest similarity to the identified user clusters within current intent.

2. Identify the recommendations from the elements of intent vector through a federated search fusion.

4.2 Decay and Trend Detection

Time factors is very important to context aware. First, all information collected is subject to time decay. This means that, for example, information from last month has less of an influence on the calculations thatn information from today. This is very important because the site may change, eg: new content added or removed, people may change their interests or concept drift happens due to new fashion. Trend detection based on time series analysis is required as a result. If a strong negative trend is detected for a given piece of content, the information associated with that content in the similarity engine is decayed at a more rapid rate. Then the likelihood of the recommendations on that piece of content is reduced. Another algorithm will also be useful for trend prediction. Eg, if a piece of content shows a strong positive trend and that same trend can be found at regular intervals in the past, then that piece of content can be automatically boosted in anticipation of the coming trend.

4.3 Browsing Behavior

Existing technologies focus on clicks, such as web link clicks. If a person clicks on a link, that click is reported. The click may be reported as having resulted in a viewed web page, even though not much time is spent on the page on which the person clicked. Thus, this known approach is not a good indication one way or another if the page is good or bad. If a link is put in a prominent position on a web page, then people are likely to click on it. However, when people get to the location indicated by the link, they may immediately leave the site. This is why the number one used button on the browser is the `Back` button. The use of the `Back` button could indicate like or dislike of a site. Accordingly, the system should recognizes that it is the action of th user after the click that matters and not the click itself. The system tracks behaviors beyond the click to determine whether a page is good or bad. Thus, if a person backs out of the page, it is considered negative feedback—which means the person did not like it. In this way, click can identify a very negative reaction. If a person goes to a link, follows the link down, spends time there, and does other things, that behavior is tracked as well. Altogether these behaviors are captured:

- Pages visited and in what order
- Time spent on each page
- Links clicked
- Searches performed
- Time spent scrolling on a page
- Portion of page visible in the browser window, and for how long

- Page sub-elements opened/closed
- Media lauched, time spend viewing, and explicit action taken on the media
- Use of the back-button
- Repeated visits to a particular piece of content
- Mouse movement while on the page
- Ads viewed and Ads clicked
- Explicit actions, including: add to cart, purchase, email. save, print, click `Like` button or `Hate` button

The only feasible approach to gather the above browsing behavior is through a `javascript` library because we could not ask site visitors to install our browser tool bar. All browsing behaviors will be reflected to three open API interfaces:

- Like
- Hate
- Visit

4.4 Semantic Similarity Measure

Our current semantic simialrity solution, no matter it's `ESA` or `Random Indexing`, all based on the fact of co-occurance. The research work in [9], which has been proven excellent on keyphrase extraction from `Sina Weibo`, caused our interests that build the term-term semantic similarity based on `Statistical Machine Translation` model—something very similar with translation language model used for faceted search, might deserve us trying because it does not depend on external knowledge such as `Wikipedia`, and could have better utility on new word discovery.

4.5 Semantic Similarity Engine

Each user is represented as a vector composed of terms extracted from user profiles and item histories. Semantic similarity measure is provided to build up `User-User`, `User-Item`, `Item-Item` linking graph with a temporal dimension. Term extraction will reuse topic detecion algorithms with a major difference—we need keywords instead of keyphrases in the recommender context situation. As shown in figure 7, the terms could in fact be looked on as tags which are used in social media community, as a result, building semantic simialarity graph based on those `tags` could be looked on as a virtual automatic folksonomy process. Additionally, given a query, we also use this similarity engine to find the user intention terms from query string. Topic(or tag) indicated search has been proven to be excellent on new items' discovery and seredenpity through our successful `TG` application. Another viewpoint on such virtual folksonomy process is: it's in fact a dimension reduction process.

The exploitation for finding user clusters could be performed through two approaches:

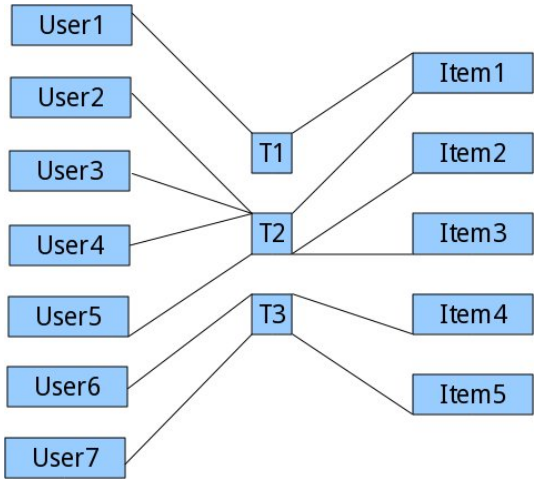

Figure 7: Topic Representation for User and Items

- Each user has been indicated as a term vector, we use pair-wise similarity join algorithm to perform the clustering process.
- If we need the clustering process to be much faster, another further dimension reduction could be performed as well—map the user vector to an integer using random algorithms, such as locality sensitive hashing ,this idea has also been adopted by Google news recommedation—[3]. Such process could be performed more incremental with a shorter periodical interval.

4.6 Learning To Rank With Exploitation and Exploration Tradeoff

Learning to rank[8] is an important research area of information retrieval, and has been adopted by both `Yahoo! Search` and `Microsoft Bing`. Existing research results have focused on introducing supervised learning past query logs as well as clickthrough data to predict the eventual ranking functions. Learning to rank should have helped a lot for recommender system because predicting top items to users is in fact a ranking problem. As mentioned above, we have in fact borrowed a lot from the research results from search engine algorithms to provide context-aware recommender. There's still one problem left partially unresolved:

- How to adjust recommendations real-time whenever user like or hate a recommendation explicitly or implicitly. Obviously, we could continue use above mentioned search solution to adjust top-N recommendations, however, it's a bit expensive since we have already got potential recommendations list, and we do not need to retrieve them repeatedly with only a few concepts' drift. We plan to introduce learning to rank mechanism here to adjust the top-N recommendations, the key idea is to make the ranking totally "dynamic"—namely, allowing it to change in response to user interactions real-time, combining diversity with high recall of all user intents.

The learning to rank framework we plan to use is different from most existing solutions, it comes from a very new direction that has combined the cutting edge research direction for `Online Learning`, `Reinforcement Learning`—learning to rank with exploitation and exploration tradeoff, which works in settings where no training data is available before deployment—it learns directly from implicit feedback inferred from user interactions. Its assumptions are somewhat more reasonable than tranditional learning to rank approaches because collecting a broad enough set of user feedback to enable effective machine learning is difficult in practice, additionally, general online feedbacks are strongly biased towards the top results, because users are unlikely to examine lower ranked documents—the new research results for both `Online Learning` and `Reinforcement Learning` shows that exploration could resolve such challenges, for example by interleaving a document list produced by the current best solutions with that of a (randomly selected) exploratory solution[13]. However, purely exploratory behavior may harm the quality of the top-N result list presented to the user. for example, once the system has found a reasonably good solution, most exploratory result lists will be worse. As a result, an exploitation-exploration trade-off is reasonable. If the system presents only top-N lists that it expects will satisfy the user, it cannot obtain feedback on other, potentially better solutions. However, if it presents document lists from which it can gain a lot of new information, it risks presenting bad results to the user during learning. Therefore, to perform optimally, the system must explore new solutions, while also maintaining satisfactory performance by exploiting existing solutions. And in our framework, we use such balance within item lists got through context-aware searching results with a re-rank using exploitation and exploration balance to provide top-N recommendations in real-time. The semantic similarity engine could be looked on as a feature selection stage for online exploration learning. To the best of our knowledge, there has not appeared a recommender framework that aims to resolve the real-time context-aware issues together with diversity, novelty and seredenpity challenges in such a mechanism—semantic search + exploration learning, and we believe its a correct direction to face the personalization challenges.

A standard mathematical setting for the exploitation and exploration balancing is `k-Armed Bandits`, often with various relevant embellishments. The k-Armed Bandit setting works on a round-by-round basis. On each round:

- A policy chooses arm a from 1 of k arms (i.e. 1 of k items).
- The world reveals the reward ra of the chosen arm (i.e. whether the ad is clicked on).

As information is accumulated over multiple rounds, a good policy might converge on a good choice of arm (i.e. items). This setting (and its variants) fails to capture a critical phenomenon: each of these items are done in the context of a situation. To model this, we might think of a different setting where on each round:

- The world announces some context information $x_t \in X$.
- A policy chooses arm a from 1 of k arms (i.e. 1 of k items) $a_t \in \{1, ..., K\}$.

- The world reveals the reward ra of the chosen arm (i.e. whether the item is liked by the user) $r_t(a_t)\epsilon[0,1]$.

So the k-Armed Bandits problem is also called `Contextual Bandits` in our situation where the goal of a learning agent is to find a policy for step 2 achieving a large expected reward—Efficiently competing with a large reference class of possible policies $\mathbb{Q} = \{\pi : X \to \{1, ..., K\}\}$:

$$Regret = \max_{\pi\epsilon\mathbb{Q}} \sum_{t=1}^{T} r_t(\pi(x_t)) - \sum_{t=1}^{T} r_t(a_t)$$

The baseline exploration strategy is $\epsilon - Greedy$ exploration—the learner selects an action with probability ϵ at each timestep. With probability $1 - \epsilon$, it selects the greedy action, i.e.,the action with the highest currently estimated value. In [4], a probability interleave algorithm is provided for learning to rank task based on $\epsilon - Greedy$, as shown in 1: the algorithm takes as input two item lists l_1 and l_2, and an exploration rate k. For each rank of the result list to be filled, the algorithm randomly picks one of the two item lists(biased by the exploration rate k). From the selected list, the highest-ranked item that is not yet in the combined result list is added at this rank. The result list is displayed to the user and clicks(likes) C are observed. Then for each liked item, a like is attributed to that list if the item is in the top N of the list.

Algorithm 1: $\epsilon - greedy$ Algorithm Variation for Learning to Rank

input : l_1, l_2, k
output: Sequence of no regret choices

initialize empty result list I;
// construct result list ;
for rank r in (1..10) **do**
  $L \leftarrow l_1$ with probability k,l_2 with probability 1 - k ;
  $I[r] \leftarrow$ first element of $L \notin I$;
end
display I and observe clicked items C ;
$N = length(C); c_1 = c_2 = 0;$
for i in (1..N) **do**
  if $C[i] \in l_1[1:N]$ **then**
    $c_1 = c_1 + 1(h,i)//countclicksonl_1$;
  if $C[i] \in l_2[1:N]$ **then**
    $c_2 = c_2 + 1(h,i)//countclicksonl_2$;
end
$n_1 = |l_1[1:N] \bigcap I[1:N]|;$
$n_2 = |l_2[1:N] \bigcap I[1:N]|;$
$c_2 = \frac{n_1}{n_2} * c_1;$
return $c_1 < c_2$

More advanced contextual bandits algorithms appear recently that have been applied to news recommendation and online targeted advertisements for `Yahoo!` [7]

[6], which aims to reduce computation cost using bandit Gradient Descent[14] under a tight regret bound, we will apply such research works to learning to rank situations as used in our context-aware recommender gradually.

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] A.S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In Proceedings of the 16th international conference on World Wide Web, pages 271–280. ACM, 2007.

[4] K. Hofmann, S. Whiteson, and M. de Rijke. Balancing exploration and exploitation in learning to rank online. Advances in Information Retrieval, pages 251–263, 2011.

[5] 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.

[6] L. Li, W. Chu, J. Langford, and R.E. Schapire. A contextual-bandit approach to personalized news article recommendation. In Proceedings of the 19th international conference on World wide web, pages 661–670. ACM, 2010.

[7] L. Li, W. Chu, J. Langford, and X. Wang. Unbiased offline evaluation of contextual-bandit-based news article recommendation algorithms. In Proceedings of the fourth ACM international conference on Web search and data mining, pages 297–306. ACM, 2011.

[8] T.Y. Liu. Learning to rank for information retrieval. Foundations and Trends in Information Retrieval, 3(3):225–331, 2009.

[9] Zhiyuan Liu., Xinxiong Chen., Yabin Zheng., and Maosong Sun. Automatic keyphrase extraction by bridging vocabulary gap. In The 15th Conference on Computational Natural Language Learning CoNLL 2011, 2011.

[10] 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.

[11] 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.

[12] A.V. Smirnov and A.A. Krizhanovsky. Information filtering based on wiki index database. In Computational intelligence in decision and control: proceedings of the 8th International FLINS Conference, Madrid, Spain, 21-24 September 2008, volume 1, page 115. World Scientific Pub Co Inc, 2008.

[13] Y. Yue, J. Broder, R. Kleinberg, and T. Joachims. The k-armed dueling bandits problem. In Conference on Learning Theory (COLT). Citeseer, 2009.

[14] Y. Yue and T. Joachims. Interactively optimizing information retrieval systems as a dueling bandits problem. In Proceedings of the 26th Annual International Conference on Machine Learning, pages 1201–1208. ACM, 2009.