A Hybrid Recommender System of Tencent Microblog

Development Document

Yingzhen Li, Ye Zhang

Abstract

In this document, we introduce the software we developed to solve the KDD Cup 2012 Track 1 problem, which required a recommender system for Tencent Microblog. We describe the data set the competition provided, the data structure and the algorithms. The mathematical proof for these algorithms is discussed in our KDD2012 paper and the project review.

1 Brief Index

This document is organized as follows. Section 2 discusses the background of the problem, and Section 3 describes the design of the hybrid recommender system. Section 4 presents the training process. Section 5 shows the experimental results and discuss some improvements, and the paper is concluded in Section 6.

2 Dataset

The dataset represents a sampled snapshot of Tencent Weibo users’ preferences for various items - the recommendation of items to users and the history of users’ ‘following’ history. It is of a larger scale compared to other publicly available datasets ever released. Also it provides richer information in multiple domains such as user profiles, social graph, item category, which may hopefully evoke deeply thoughtful ideas and methodology.

To protect the privacy of the users, the IDs of both the users and the recommended items are anonymized as random numbers such that no identification is revealed.

2.1 Item

An item is a specific user in Tencent Weibo, which can be a person, an organization, or a group, that was selected and recommended to other users. The size of this is about 6K items in the dataset.

2.1.1 Item Hierarchy

Items are organized in categories; each category belongs to another category, and all together they form a hierarchy. For example, an item, a vip user Dr. Kaifu LEE, is represented as science-and-technology.internet.mobile (see Figure 1).

We can see that categories in different levels are separated by a dot ‘.’, and the category information about an item can help enhance your model prediction.
2.1.2 **Item data: item.txt**

Format: \( (ItemId) \langle Item - Category \rangle \langle Item - Keyword \rangle \)

Item-Category is a string 'a.b.c.d', where the categories in the hierarchy are delimited by the character '.', ordered in top-down fashion (i.e., category 'a' is a parent category of 'b', and category 'b' is a parent category of 'c', and so on.

Item-Keyword contains the keywords extracted from the corresponding Weibo profile of the person, organization, or group. The format is a string 'id1;id2;...;idN', where each unique keyword is encoded as an unique integer such that no real term is revealed.

2.2 **User**

2.2.1 **User profile data: user_profile.txt**

Format: \( (UserId) \langle Year - of - birth \rangle \langle Gender \rangle \langle Number - of - tweet \rangle \langle Tag - Ids \rangle \)

Year of birth is selected by user when he/she registered.

Gender has an integer value of 0, 1, or 2, which represents 'unknown', 'male', or 'female', respectively.

Number-of-tweet is an integer that represents the amount of tweets the user has posted.

Tags are selected by users to represent their interests. If a user likes mountain climbing and swimming, he/she may select "mountain climbing" or "swimming" to be his/her tag. There are some users who select nothing. The original tags in natural languages are not used here, each unique tag is encoded as an unique integer.

Tag-Ids are in the form 'tag-id1;tag-id2;...;tag-idN'. If a user doesn’t have tags, Tag-Ids will be '0'.

2.2.2 **User action data: user_action.txt**

Format: \( (UserId) \langle Action - Destination - UserId \rangle \langle Number - of - at - action \rangle \langle Number - of - retweet \rangle \langle Number - of - comment \rangle \)

If user A wants to notify another user about his/her tweet/retweet/comment, he/she would use an 'at' (@) action to notify the other user, such as '@tiger' (here the user to be notified is 'tiger').

For example, user A has retweeted user B 5 times, has 'at' B 3 times, and has commented user B 6 times, then there is one line 'A B 3 5 6' in user_action.txt.
2.2.3 User sns data: user_sns.txt

The file user_sns.txt contains each user’s follow history (i.e., the history of following another user). Note that the following relationship can be reciprocal.
Format: (Follower – userid) t(Followee – userid)

2.2.4 User key word data: user_key_word.txt

Format: (UserId) t(Keywords)
Keywords is in the form kw1 : w1; kw2 : w2; …kw3 : w3.
Keywords are extracted from the tweet/retweet/comment of a user, and can be used as features to better represent the user in your prediction model. The greater the weight, the more interested the user is with regards to the keyword.
Every keyword is encoded as a unique integer, and the keywords of the users are from the same vocabulary as the Item-Keyword.

2.3 Actions on Microblog

‘Tweet’: a ‘tweet’ is the action of a user posting a message to the microblog system, or the posted message itself. So when one user is ‘tweeting’, his/her followers will see the ‘tweet’.
‘Retweet’: a user can repost a tweet and append some comments (or do nothing), to share it with more people (my followers).
‘Comment’: a user can add some comments to a tweet. The contents of the comments will not be automatically pushed to his/her followers as ‘tweeting’ or ‘retweeting’, but will appear at the ‘comment history’ of the commented tweet.
‘Followee/follower’: If User B is followed by User A, B is a followee to A, and A is a follower to B.

2.4 Training Set and Test Set

Training dataset: some fields are in the file rec_log_train.txt
Testing dataset: some fields are in the file rec_log_test.txt
Format of the above 2 files: (UserId) t(ItemId) t(Result) t(Unix – timestamp)
Result: values are 1 or -1, where 1 represents the user UserId accepts the recommendation of item ItemId and follows it (i.e., adds it to his/her social network), and -1 represents the user rejects the recommended item.
We provide the true values of the ‘Result’ field in rec_log_train.txt, whereas in rec_log_test.txt, the true values of the ‘Result’ field are withheld (for simplicity, in the file they are always 0). Another difference from rec_log_test.txt to rec_log_train.txt is that repeated recommended (UserId,ItemId) pairs were removed.

3 User Guide

Please put the files(*.txt) of track1 dataset, rec_log_test.txt, sub_small_header.csv and the codes in the same directory, and run the codes in that directory as follows:
- run DATAIMPORTRAW.py: loading data
- run Distributed_Apriori.py: keyword analysis
- actually you can run it on a single core, if you want ro distribute it, please install mpi modules (mpich2, openmpi, etc.) and python module mpi4py, then type in the terminal as mpirun –n #core python DistributedApriori.py (&)(> your_log_name)

- run CLASS_CATE.py: keyword analysis
- run STATISTICS.py: user taxonomy
- if you want to use cross-validation method, please run split.py for separating original training set, and rename the 2 output file as data_rec_log_train and data_rec_log_test
- run TRAIN_PREFERENCE.py: training and prediction, saved in data_output_evaluation
  - parameters embeded in each file is changable, directly change it to see the results of different set of coefficients;
- to test the algorithm directly on the original test set, please run train_prediction.py and upload the generated csv file on the official website, and get the MAP@3 of the recommendation

NOTICE: Since we lost the computing support before we test our algorithm on the whole set, we are not sure the code could work stably.

4 Experiment

4.1 Training Result

In our experiment we sampled 5,938 users’ recommendation records from the dataset stochastically and divided them into 2 subsets for training and testing. We assigned the same proportion of at(®), retweet and comment when computing fami(uj; uk). Table 1 presents the results of the training process. The result shows an evident discrepancy of α1, which reflects the inclination of accepting popular items. Inactive users prefer items with similar interests while active users prefer items with high popularity.

<table>
<thead>
<tr>
<th>user class</th>
<th>user</th>
<th>followee</th>
<th>interaction</th>
<th>keyword</th>
<th>α1</th>
</tr>
</thead>
<tbody>
<tr>
<td>active</td>
<td>3919</td>
<td>46</td>
<td>87</td>
<td>10</td>
<td>0.33</td>
</tr>
<tr>
<td>inactive</td>
<td>1194</td>
<td>27</td>
<td>42</td>
<td>6</td>
<td>0.18</td>
</tr>
<tr>
<td>fake</td>
<td>825</td>
<td>18</td>
<td>2</td>
<td>5</td>
<td>/</td>
</tr>
</tbody>
</table>

Table 1: Training Sets and Optimal Parameters. Fake users’ grading function has no parameters to update so we omit the training process of it.

4.2 Prediction and Precision Evaluation

We computed grade(uj, ik) of all result(uj, ik) in testing subset and generated ordered item list of uj (see section 3.3.4) to test the trained system. The evaluation metric is the mean average precision AP@3 which KDD Cup’s organizers adopted. Table 2 presents the MAP@3 (mean value of AP@3(uj)) results and Table 3 presents the recommended item lists and the average precision of some users. The precision of fake users’ prediction is much lower than others’ in our experiment due to the difficulty of their interests’ extractions. Adjusting min_action or recommending their linkers on other related platforms like QQ might help improve the results.
Table 2: Prediction Evaluation. Mining potential interests from inactive users’ followees improves the performance of recommendation. Fake users’ result is not good as the others.

<table>
<thead>
<tr>
<th>active</th>
<th>inactive</th>
<th>fake</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.41066</td>
<td>0.46879</td>
<td>0.33606</td>
<td>0.41198</td>
</tr>
</tbody>
</table>

Table 3: Examples of Prediction. User 2071402 accepts the 1st and 3rd items, then $AP_{@3} = \frac{1 + 3}{2} = \frac{5}{2}$; User 942226 only accepts the 1st item, then $AP_{@3} = \frac{1}{1} = 1$; User 193889 only accepts the 3rd item, then $AP_{@3} = \frac{1}{3}$.

A Appendix: Data Structure and Classes

We developed the system by revising the structure of the data, described as follows. The description was followed the formate of Python.

A.1 Keyword Class

- list keyword_class = [[kw1,...],...]
  - generated by the FDM Algorithm 1 (Distributed_Apriori(data_user_key_word))
  - get_class(key): return the #class of the keyword key

- dict class_cate = {class1 : {cate1 : w1,...},...}
  - generated by
  - denote the keyword classes belonging to the category (hierarchy) cate i

A.2 Item

- dict itemCate = {item1 : cate1,...}
  - get_rank(item, cate): return bool values

- dict cateItem = {cate1 : [item1,...],...}

- dict itemKey = {item1 : {kw1 : w1,...},...}
  - get_item_key(itemId): return the keywords of the item itemId
  - get_key_item(key): return the items who own the keyword key
A.3 User

- `dict data_user.profile = {userId: [#age, gender, #tweet]}`
- `dict data_user.action = {user1: {targetUser1: [#at, #retweet, #comment],...},...}`
  - `get_action(userId, targetUserId): return the interactions between user userId and targetUserId`
- `dict data_user.sns = {user1: [followee1,...],...}`
  - `get_followee(userId): return the followees’ Id of the user userId`
- `dict data_user.key_word = {user1: {kw1: w1,...},...}`
  - `with the same methods of itemKey`
- `dict data_user.key_class = {user1: {class1: w1,...},...}`
  - `generated from data_user.key_word and keyword_class`

B Appendix: Algorithms

**Algorithm 1 Distributed_Apriori(data_user.key_word)**

1: `home: j = 1`
2: `home: divide DB into DB_i`
3: `home: send DB_i to RM_i`
4: `while T^j is updated do`
5:  `if j > 1 then`
6:   `home: broadcasts the transactions T_i^j to the corresponding remote sites`
7:  `end if`
8:  `for each remote site RM_i do`
9:   `gather T_i^j from home`
10:  `C_i^j = Apriori_gen(T_i^j^{-1})`
11:  `eliminates C_i^j if fail to satisfy supp_local or conf_local`
12:  `send C_i^j to PLK, K = polling(C_i^j)`
13:  `end for`
14:  `for each polling site PLK do`
15:   `gather C_i^j from remote sites`
16:   `broadcast requests for supp_local to remote sites`
17:   `gather supp_local from remote sites`
18:   `eliminates C_i^j if fail to satisfy supp_global or conf_global`
19:   `send C_i^j to home`
20:  `end for`
21:  `home: gather C_i^j from polling sites`
22:  `T^j = \bigcup_i C_i^j`
23: `end while`
24: `return keyword_class = T^j`
Algorithm 2 $user.class(u_j)$

1: initialize \#at, \#retweet, \#comment as 0  
2: $min\text{-}activeness = 100$, $min\text{-}action = 20$  
3: for each $targetUserId$ in $data.user.action[u_j]$ do  
4: \#at, \#retweet, \#comment + = $get\_action(userId,targetUserId)$  
5: end for  
6: $ACT = act(u_j) = tweet \times is\_fake(u_j)$  
7: if $ACT \geq min\_activeness$ then  
8: CLASS = 2 //active  
9: else  
10: if $0 < ACT < min\_activeness$ then  
11: CLASS = 1 //inactive  
12: end if  
13: else  
14: CLASS = 0 //fake  
15: end if  
16: return CLASS

Algorithm 3 $get\_hot\_rank(i_k, h_k)$

1: if $rank$ not initialized then  
2: initialize dict $rank[h_k] = \{i_1 : rank_1\}$  
3: for each item $i_j$ in $h_k$ do  
4: from $rec\_log\_train.txt$ get all recommendation records of $i_j$  
5: for each recommendation record do  
6: if recommendation accepted then  
7: $rank[h_k][j]+ = 1$  
8: end if  
9: end for  
10: $rank[h_k][j] = \frac{2}{1+exp^{-rank[j]}}$  
11: end for  
12: sort $rank[h_k]$ by descendent  
13: save $rank[h_k]$  
14: end if  
15: return $rank[h_k][i_k]$
Algorithm 4 $GET\_HOT\_RANK(i_k)$

1: if $RANK$ not initialized then  
2:    initialize dict $RANK = \{i_1 : rank_1\}$  
3:    for each item $i_j$ in $I$ do  
4:        from rec_log_train.txt get all recommendation records of $i_j$  
5:        for each recommendation record do  
6:            if recommendation accepted then  
7:                $RANK[j] += 1$  
8:            end if  
9:        end for  
10:       $RANK[j] = \frac{2}{1+exp^{\text{rank}[j]}}$  
11:    end for  
12:    sort $RANK$ by descend  
13:    save $RANK$  
14: end if  
15: return $RANK[i_k]$

Algorithm 5 $get\_user\_key\_class(u_j, data\_user\_key\_word, keyword\_class)$

1: initialize dict $user\_key\_class = \emptyset$  
2: $K_j = get\_user\_key\_word(u_j, data\_user\_key\_word)$  
3: for each keyword class $class_i$ do  
4:    for each keyword $k_l$ in $K_j$ do  
5:        if $k_l \in class_i$ then  
6:            if $class_i \notin user\_key\_class$ then  
7:                add $class_i$ into $user\_key\_class$  
8:            assign its weight $W_i = w_l$  
9:            else  
10:           $W_i += w_l$  
11:        end if  
12:    end if  
13: end for  
14: end for  
15: return $user\_key\_class$

Algorithm 6 $potential\_key(u_j, key\_class)$

1: initialize dict $POTENTIAL\_KEY = \emptyset$  
2: $related\_users = search\_followee(u_j, depth)$  
3: for each $u_k$ in $related\_users$ do  
4:    for each keyword class $class_i$ in $key\_class$ do  
5:        if $class_i$ not in $POTENTIAL$ then  
6:            add $class_i$ into $POTENTIAL$  
7:            $W_{ji} = W_{kl} fami(u_j, u_k)$  
8:        else  
9:           $W_{ji} += W_{kl} fami(u_j, u_k)$  
10:       end if  
11:    end for  
12: end for  
13: return $POTENTIAL\_KEY$
Algorithm 7 \textit{interests}(u_j)

1: initialize $\text{KEY\_CLASS} = \text{get\_user\_key\_class}(u_j, \text{data\_user\_key\_word}, \text{keyword\_class})$
2: initialize $\text{POTENTIAL\_KEY} = \text{potential\_key}(u_j, \text{key\_class})$
3: initialize dict \text{INTERESTS} = \emptyset
4: for each possible class $i$ do
5: \hspace{1em} if $i \in \text{KEY\_CLASS}$ then
6: \hspace{2em} add $i$ into \text{INTERESTS}
7: \hspace{2em} assign corresponding weight $W_i = \overline{W}_i$
8: \hspace{1em} end if
9: \hspace{1em} if $i \in \text{POTENTIAL\_KEY}$ then
10: \hspace{2em} if $i \in \text{INTERESTS}$ then
11: \hspace{3em} $W_i = \frac{W_i + \overline{W}_i}{2}$
12: \hspace{2em} else
13: \hspace{3em} add $i$ into \text{INTERESTS}
14: \hspace{3em} assign corresponding weight $W_i = \overline{W}_i$
15: \hspace{2em} end if
16: \hspace{1em} end if
17: end for
18: return \text{INTERESTS}

Algorithm 8 \textit{get\_category}(\text{user\_key\_class}, \text{class\_cate})

1: initialize dict \text{CATEGORY} = \emptyset
2: for each category $h_k$ do
3: \hspace{1em} for each keyword class $i$ in \text{user\_key\_class} do
4: \hspace{2em} if $i \in \text{KH}(h_k)$ then
5: \hspace{3em} add $h_k$ into \text{CATEGORY}
6: \hspace{2em} end if
7: \hspace{1em} end for
8: end for

Algorithm 9 \textit{get\_similarity}(u_j, i_k)

1: initialize $\text{similarity} = 0$
2: for each $i$ in \text{user\_key\_class}(u_j) do
3: \hspace{1em} if $i \in \text{user\_key\_class}(i_k)$ then
4: \hspace{2em} \text{similarity} += $\overline{W}_i(u_j) \times \overline{W}_i(i_k)$
5: \hspace{2em} end if
6: \hspace{1em} end for
7: return $\frac{2}{1 + \exp(\sqrt{\text{similarity}})}$

Algorithm 10 \textit{fond}(u_j, h_k)

1: initialize $\text{FOND} = 0$
2: for each $i$ in \text{interest}(u_j) do
3: \hspace{1em} if $i \in \text{KH}(h_k)$ then
4: \hspace{2em} $\text{FOND} += W_i \times \overline{W}_i$
5: \hspace{2em} end if
6: \hspace{1em} end for
7: return $g(\text{FOND}, 100)$
Algorithm 11 grade($u_j, i_k$)

1: initialize $W = 0$
2: $userClass = user\_class(u_j)$
3: $INTERESTS = interest(u_j)$
4: $CATEGORY = get\_category(INTEREST, class\_cate)$
5: if $i_k$ in $h_k \in CATEGORY$ then
6: $W = fond(u_j, h_k)$
7: end if
8: if userClass > 0 then
9: $hot = get\_hot\_rank(i_k, itemCate, cateItem)$
10: $similarity = get\_similarity(u_j, i_k)$
11: $GRADE = 2W(\alpha_1 hot + \alpha_2 similarity) - 1$
12: else
13: $HOT = GET\_HOT\_RANK(i_k)$
14: $GRADE = (1 + W)HOT - 1$
15: end if
16: return $GRADE$