SimpleWalk: RECSYS2016

20120298 CSE 김동석

waegaein@postech.ac.kr

- Table of Contents -

I. Algorithm

I.1. Design

1. Simple solution generated by impressions data of most recent week scored over 300,000 points. (Quite big!)

   Also, impressions data is the actual previous recommendations.

→ **Start on the top of impressions data would be good choice.**

2. Simple solution generated by impressions data of most recent week sampled scored more about 10 times than that generated by impressions data of least recent week sampled.

→ **Use the impressions data of most recent week.**

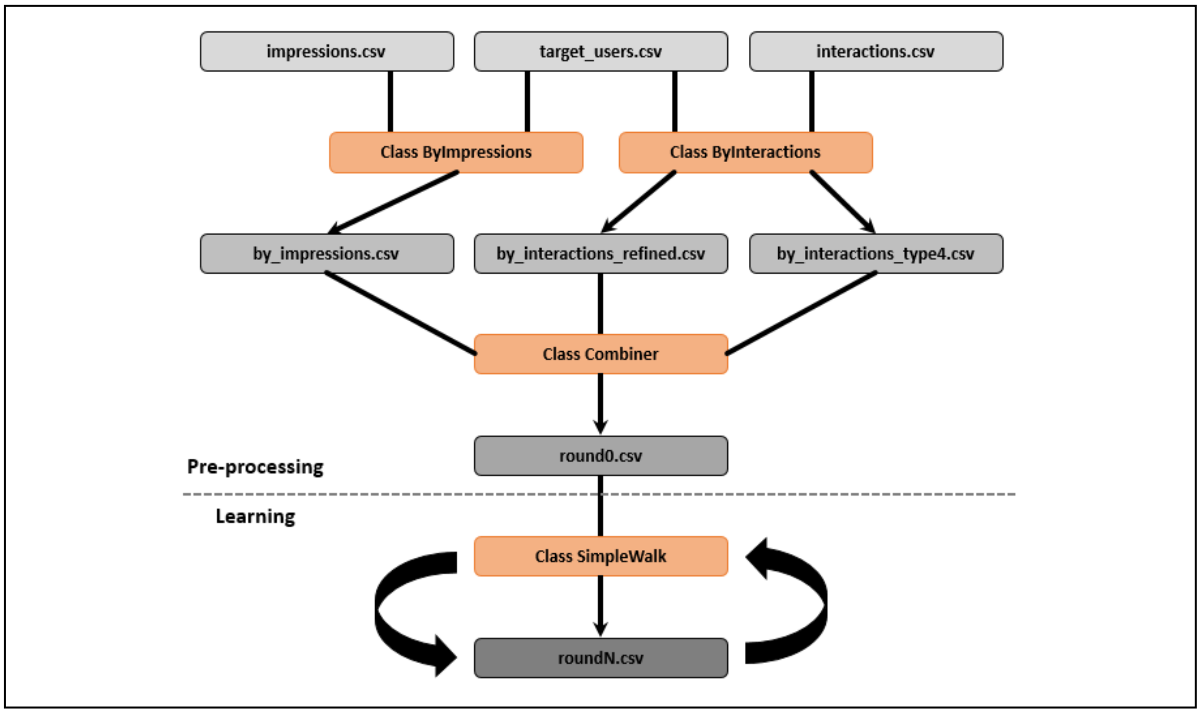
3. Simple solution generated by refined interactions data, which picks of which type is highest other than 4 among interactions of the same user and the same item, scored more about 10% than that generated by intact interactions data. Also, type 4 is user's negative action to an item, according to the data description.

→ **Interactions of which type is 4 can be used as negative data and the rest can be used as positive data.**

4. The solution is graded up to 30 items each. Through analyzing, the impressions data showed the average number of items about 21, when most recent sampled and cut up to 30 items.

→ **There remains quite many chances to enhance result by spreading already filled data to sparse ones by measuring similarity**

I.2. Concept



I.3. Pre-processing

Class ByImpressions:

Among all the impression data, take ones about the target users. If there are multiple data about thesame user, take one with the most recent week.

Class ByInteractions:

Among all the interaction data, take ones about the target users. If there are multiple data about the same target user and item, take one with the highest type other than 4. Meanwhile, take ones with type 4 for the target users.

Class Combiner:

Among most recent impression data about target users, for each user's items, eliminate ones that is type 4 in interaction data. This is because the interaction type 4 is user's rejection to an item. After, append each user's items with the refined interaction data.

I.4. Learning

Class SimpleWalk:

For each user in the pre-processed data, find another user which has the highest similarity, above threshold value, which is measured by the item vector's similarity. After, append the user's items with the items that it doesn't' have and the most similar user has. Adjusting or maintaining the threshold, repeat this step.

II. Hyper-parameters

Number of round:

In learning phase, the result size is incremented in each round. However, the result accuracy may increase or decrease in each round. The number of rounds determines when to stop repetition.

Threshold value (0 to 1):

In learning phase, the similarity is measured and compared to threshold value in order to determine whether to append new items to existing items. If the threshold is too low, accuracy may be too low. If the threshold is too high, the learning may go too slow. The threshold value should be adjusted through experiment or cross-validation in order to get adequate learning speed and accuracy.

Reduce condition:

In learning phase, the user with the highest similarity, above threshold value, is to be appended. But it is also the case where there exists a user with slight lower similarity (reduced similarity) has more items to be appended than the user with the highest similarity. This is determined by the reduce condition that how much to allow the 'slight lower' similarity (tolerance ratio) and how much to necessitate the 'more' items (extension ratio). If the model allows too much lower similarity, the items appended may be with low accuracy. If the model necessitates too much more items, quite good candidates may be missed. The reduce condition should be adjusted through experiment or cross-validation not to miss good candidates, without loss of accuracy.

Threshold change between rounds:

In learning phase, threshold value may be changed between rounds. If one round ends, the user whose items are appended may show much higher similarity with the user which is used to append. Then the next round may need higher or lower threshold value depending on result. Whether to and how much to change threshold between rounds should be determined through experiment of cross-validation to get higher accuracy.

III. Implementation

public class ByImpressions

class Impression:

Stores userID, year, week, items of impression data as integers.

ArrayList<Integer> userList:

Stores the target userIDs' as list of integer.

ArrayList<Impression> impList:

Stores the impression data as list of class Impression.

public void readUsers ( ):

Read a file 'target_users.csv', with memory-mapped file. Result is stored in the userList.

public void readMostRecent ( ):

Read a file 'impressions.csv'. While reading, stores the most recent week impression data for the same userID. Result is stored in impList.

public void pickTarget ( ):

Traversing impList, pick the impression data whose userID is contained in the userList, using function 'isTarget'. Result overwrites the impList.

public boolean isTarget (Impression imp):

Traversing userList, return true if there is a userID which equals to the userID of the argument imp. Unless, return false.

public void writeByImpressions ( ):

Write the impList to a file 'by_impressions.csv".

public class ByInteractions

class Interaction:

Stores userID, item, type, created time of interaction data as integers.

class PerUser:

Stores the userID and its list of interactions.

ArrayList<Integer> userList:

Stores the target userIDs' as list of integer.

ArrayList<Interaction> interList:

Store the interaction data as list of class Interaction.

ArrayList<PerUser> perUserList:

Store the interaction data grouped by their userID as list of class PerUser.

ArrayList<PerUser> refinedList:

Store the refined interaction data grouped by their useIDr as list of class PerUser.

ArrayList<PerUser> type4List:

Store the type 4 interaction data grouped by their userID as list of class PerUser.

public void readUsers ( ):

Read a file 'target_users.csv', with memory-mapped file. Result is stored in the userList.

public void readInteractions ( ):

Read a file 'interactions.csv', with memory-mapped file. Result is stored in the interList.

private void group ( ):

Traversing interList, group the interactions by their userID as the class PerUser. Result is stored in perUserList.

private void process ( ):

Traversing perUserList, pick the impression data whose userID is contained in the userList, using function 'isTarget'. Result is stored in both the refinedList and type4List. After, traversing refinedList, invoke PerUser's function 'refine' for each element. Then, Traversing type4List, invoke PerUser's function 'type4' for each element.

private boolean isTarget (PerUser pu):

Traversing userList, return true if there is a userID which equals to the userID of the argument pu. Unless, return false.

PerUser: public void refine ( ):

First, create temp, a new empty list of interactions. Traversing its original list of interactions, for the interactions of the same item, add ones with the highest type to the temp. After, temp consists of one interaction for one item, with the highest type for each item. Second, create tempNo4, a new empty list of interactions. Traversing temp, adds one whose type does not equal 4 to the tempNo4. After, tempNo4 consists of one interaction for one item, with the highest type for each item other than 4. Finally, the tempNo4 overwrites the refineList.

PerUser: public void type4 ( ):

First, create temp, a new empty list of interactions. Traversing its original list of interactions, for the interactions of the same item, add ones with the highest type to the temp. After, temp consists of one interaction for one item, with the highest type for each item. Second, create tempType4, a new empty list of interactions. Traversing temp, adds one whose type equals 4 to the tempType4. After, tempType4 consists of one interaction for one item whose type equals 4. Finally, the tempType4 overwrites the type4List.

private void writeByInteractions ( ):

Write the refinedList to a file ‘by_interactions_refined.csv”. Write the type4List to a file ‘by_interactions_type4.csv”.

public class Combiner

class Data:

Stores userID as integer and items as a list of integers.

ArrayList<Data> impList:

Stores the impression data of most recent week for target userIDs, converted to class Data.

ArrayList<Data> refinedList:

Stores the interaction data of highest type other than 4 for target userIDs, converted to class Data.

ArrayList<Data> type4List:

Stores the interaction data of type 4 for target userIDs, converted to class Data.

private void readByImpressions ( ):

Read a file ‘by_impressions.csv’, with memory-mapped file. Result is stored in the impList.

private void readByInteractionsRefined ( ):

Read a file ‘by_impressions_refined.csv’, with memory-mapped file. Result is stored in the refinedList.

private void readByInteractionsType4 ( ):

Read a file ‘by_impressions_type4.csv’, with memory-mapped file. Result is stored in the type4List.

private void process ( ):

First, traversing the impList, eliminate the items that appears in the type4List with current userID, using function ‘killFour’. Second, traversing the impList again, append the items with the items that appears in the refinedList with current userID, using function ‘append’. Third, traversing the refinedList, append the data, whose userID is not included in impList, to the impList using function ‘existAlready’. Finally, sort the impList ascending by userID.

private void killFour (Data target):

Traversing the type4List, find the data of which userID equals the userID of the argument data. If it is found, create temp, a new list of integers. Then, traversing the items of the argument data, add items, which is not included in the found data, to the temp using function ‘existFour’. After, the temp consists of the items included in the argument data’s items and not in the found data’s items. Finally, the temp overwrites the argument data’s item.

private boolean existFour (int item, ArrayList<Integer> four):

Traversing the argument four, return true if there is a userID which equals to the argument item. Unless, return false.

private void append (Data target):

Traversing the refinedList, find the data of which userID equals userID of the argument target. If it is found, create currentItems, a new list of integers consists of the found data's items. Then, append currentItems to the items of the argument target.

private boolean existAlready (Data target):

Traversing the impList, return true if there is a userID which equals to the userID of the argument target. Unless, return false;

private void writeBaseline ( ):

Write the impList to a file 'round0.csv".

public class SimpleWalk ( )

class Data:

Stores userID as integer and items as a list of integers.

ArrayList<Data> prevList:

Stores the base data.

ArrayList<Data> pickList:

Stores the data picked for the highest similarity, above threshold, to the base data.

ArrayList<Data> nextList:

Stores the revised data.

private void readPrevious ( ):

Read a file 'roundN.csv', with memory-mapped file. Result is stored in the prevList.

private void pick ( ):

Traversing the prevList, fill the picklist with the data of the highest similarity, above threshold value, to each data using function 'findSimilar'.

private void findSimilar (Data target):

First, create similar, a new data to hold the data found so far to have the maximum similarity. Next, Traversing the prevList, calculate similarity score using function 'calculateScore'. If the score is above threshold value and so far maximum, update the similar and so far maximum with the current data. Here, it is a chance that there exist quite good candidates, with more items to enhance the original data, may be missed due to the small gap of the similarity score. To take into consider this, if the score is above threshold but not so far maximum, check more. If the score is higher than 90% of so far maximum and the number of items exceed 1.1 times the number of items of the current similar, update the similar and so far maximum with the current data.

private double calculateScore (Data target, Data candidate):

If the number of items of the argument target is zero, it is impossible to calculate similarity, thus return zero. Unless, traversing the items of the argument candidate, count the number of items which exist in the items of the argument target. After, return the existing count divided by the number of items of the argument target. This measures the ratio that the argument target's items are included in the argument candidate's items.

private void process ( ):

Traversing the prevList and the pickList simultaneously with the same index, fill nextList of the index with the merged data. If the current entry in the pickList is empty, it is the case where there was no data of similar enough to the current entry. Then fill nextList with just the current entry in the prevList. Unless, fill nextList with the current entry in the prevList, of which items are merged with the items of the current entry in pickList.

private void writeNext ( ):

Write the nextList to a file 'roundN+1.csv".

Result

Solution	Round	Threshold	Reduce Condition	Size (KB)	Score	Score / Size
0	0	-	-	24,708	376,391.30	**15.23**
1	1	0.25	A	29,750	**384,302.71**	12.92
2	1	0.50	A	26,868	380,187.13	14.15
3	1	0.50	B	26,603	380,857.54	**14.32**
4	2	0.25 / 0.25	B	28,940	**384,140.53**	13.27
5	2	0.25 / 0.50	B	28,887	384,117.19	13.30
6	3	0.50 / 0.50 / 0.50	B	26,692	381,031.62	14.28

(Reduce Condition. A: tolerance ratio 0.9, extension ratio 1.0 / B: tolerance ratio 0.9, extension ratio 1.1)

The solution 1 and 2 shows that the lower threshold makes more append operations and thus increase both size and score more. Meanwhile, the score/size ratio of them tells that there is a significant loss of accuracy when we user threshold value of 0.25. Compared to the solution 0, the gap of the solution 1 is about 2 times of that of the solution 2. The solution 2 and 3 shows that the reduce condition B, with higher extension ratio, makes findSimilar operation stricter and then achieve higher accuracy.

Discussion

The training phase took about 3 hours for a round and score evaluation was possible 5 times per day. So the meaningful examples evaluated was just 6, thus I failed to find the optimal hyper-parameters. With more time and faster machine or better algorithm, it would be possible to test various conditions and get better result. Besides, there is limitation of the SimpleWalk. It started on the top of existing recommendation and spread them among users based on similarity. It cannot generate recommendation to users not included in impressions data. The recommendation for the users started with very small items may go biased during progress. Using SimpleWalk with the other methods deals with the limitations may work far better.

Environment

Language:

java version "1.8.0_73"

Java(TM) SE Runtime Environment (build 1.8.0_73-b02)

Java HotSpot(TM) 64-Bit Server VM (build 25.73-b02, mixed mode)

IDE: Eclipse (Version Mars.2 Release 4.5.2)

Memory: 24GB

CPU: i5-6600 (4 cores, 3.30GHz)