

**A Functional-Logic Model of Collaborative Filtering with Slope One
Recommendations in eTourism**

CS6715 Project Proposal

Saturday, March 31, 2007

Tshering Dema, Shifa Sharif, Greg Sherman

Supervised by: Dr. Harold Boley

Abstract

In eCommerce, eTourism, and in other electronic ventures, it is becoming increasingly important to recommend offerings to prospective travelers on an individual basis. By collecting preferences from many different common sources, we engage in a process known as collaborative filtering. Various methods of predicting what a consumer would like to see have been developed thus far, some very accurate, yet prohibitively complex. There are others, which in sacrificing some accuracy in prediction become easy to implement, and understand. One such method is the use of the Slope One algorithm[1]. A prediction using the Slope One method ($f(x) = x + b$, where b is a constant and x is a variable representing rating values) will use data provided by other users to ‘interpolate’ a value approximating the user’s preference. In addition to recommendations based on such subjective ratings, we can also use rules for recommending destinations based on objective characteristics, thus blocking or boosting ratings for the overall recommendation of a destination.

This method is currently being used in several systems. The first example of its implementation is the NRC project inDiscover[2], now jointly with Bell/MSN. This is a service that allows users to rate different musical compositions, and by using this data, make recommendations to other users that have previously rated common pieces of music.

Many implementations of this method have been developed. The aforementioned project, inDiscover, made use of a Java implementation. There is a very simple and easy to understand Python implementation available online as well[3]. To demonstrate the diverse ways in which this method can be invoked, we intend on implementing a Slope One algorithm in the realm of eTourism using the functional-logical language RELFUN[4].

We envision a collaborative system wherein tourist destinations are rated on a scale of one to five (1-5), with separate ratings for many aspects that are sought after in such a destination (“multidimensional” ratings). Such ratings would include overall ratings (possibly derived from sub-ratings, or given as an unrelated overall impression), ratings based on service, accommodations, climate, activities and so on.

Our implementation of the Slope One prediction scheme is meant only to be a demonstration, and as such we will not be concerned with storing new ratings, or changing the ratings we have already recorded. For simplicity, we will choose not to derive the overall rating from the subratings.

Solution Approach

- Build the database
 - Our database is statically defined
 - Uses ground facts such as rating(),user(),destination(),climate(), subclassOf(), etc.
- Implement the Slope One Algorithm in RELFUN
 - Generate a user specific table of deviation values, stored in a list using RELFUN's tupof() primitive.
 - Predict a user's rating for each destination for which the user wishes to see a recommendation (e.g. "Predict my rating for all European destinations").
 - Store these predictions in a list, which may later be modified according to user preferences (see next step in Solution Approach).
- Implement rules testing objective characteristics for blocking or boosting predictions. Examples of applying additional user-specified rules to modify the results that will be predicted:
 - If user Bob knows that user Fred's tastes are polar opposites of his own, Fred's ratings might be inverted, or, more cautiously, not be considered.
 - If Bob does not like climates where the average temperature is above 40° C, such destinations will not be recommended to him.
 - If Bob considers himself an Asian history aficionado, he knows that he will like Asian countries more, so we would apply a boost to Asian destination over all others.

A Simple Slope One Example

Consider we have the following travel destination ratings given by Bob, Fred and Jack.

	Burma	Tokyo	Thimpu	Delhi
Bob	4	–	3	4
Fred	2	–	1	–
Jack	–	3	–	5

We can represent these as ground facts in RELFUN in the following manner:

```
rating(User, Destination, Rating_Value).
```

```
rating(jack, tokyo, 3).
```

```
rating(jack, delhi, 5).
```

```
rating(fred, burma, 2).
```

```
rating(fred, thimpu, 1).
```

```
rating(bob, burma, 4).
```

```
rating(bob, thimpu, 3).
```

```
rating(bob, delhi, 4).
```

If we would like to predict how Fred would rate Delhi, we first need to generate a collection of values that represent the average deviations of user's ratings. We are only concerned with looking at ratings from people who have rated destinations in common with Fred. In this case Jack has not rated any destinations which Fred has also rated, so we will disregard all of Jack's ratings.

```
%we consider only overall rating here
```

```
%for each destination that Fred has rated,
```

```
%make a list of all other users that have rated those
```

```
%destinations. Remove duplicates from the list.
```

```
getList([]):&[].
```

```
getList([Did|T]):- L .= tupof(rating(Uid, Did, overall,  
R), [Uid, Did, R]) &concate(L, getList(T)).
```

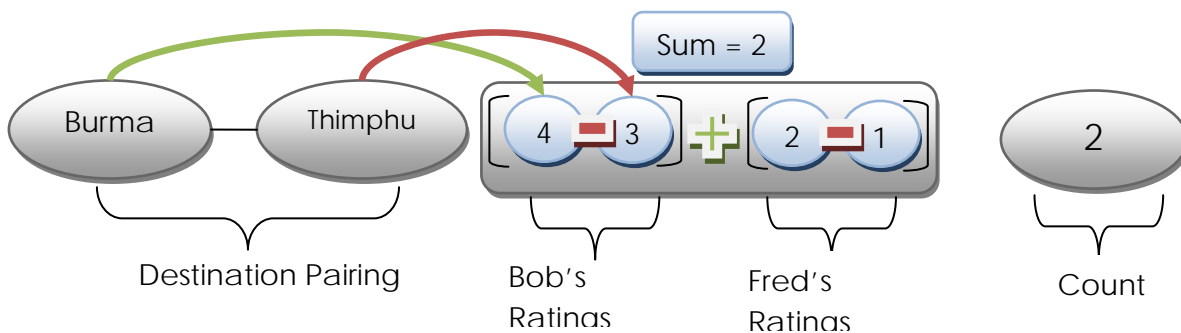
A deviation value is a representation of pairs of items that have been rated by either the person for whom we want to predict a rating (in our case, Fred), and persons who have rated at least one destination in common with this person. For this example, the only other person who has rated destinations in common with Fred is Bob. They have both given ratings for Burma and Thimpu.

We must first create a list of destination pairs. We could now exclude destinations in which Fred has no interest. For now, we will avoid the complexity of applying such constraints. This list will look as follows:

```
[Burma,Thimpu],[Thimpu,Delhi],[Thimpu,Burma],[Burma,Delhi]
```

Each deviation value consists of a destination pairing, a sum of differences of ratings, and a count. When considering the differences of ratings, the order of the difference matches the order of the destination pairing. Count is the number of users who have rated this pair of destinations.

Example:



Here we also need user specific list of ratings which can be generated as follows:

```
getUserSpecificFacts(Uid):-
%Rated only by Uid, we are intersted about these Destinations
Struct1 .= tupof(rating(X, Uid, Did, overall, R),Did),
Struct2 .= getList(Struct1)
&Struct2.
```

Input/Output in Relfun

```
rfi-p> getUserSpecificFacts(user1)
```

```
[[user1,dest03,4],
```

```
 [user0,dest03,4],
```

```

[user2,dest03,4],
[user2,dest04,3],
[user0,dest04,3],
[user1,dest04,5],
[user0,dest01,2],
[user1,dest01,2]]

```

Given a pair of destination and a list of user specific facts the sum and count for this pair can be generated as follows:

```

check([_, Did1, _],Did1,Did2):&1.
check([_, Did2, _],Did1,Did2):&2.
check([_, X, _],Did1,Did2):&0.

```

%base Case

```

checkUser(Tuple, [], Did, Sm, Cnt):- X .= cns2(Sm,Cnt) &X.

```

%recursive Case - current fact matches user Id (Uid) of given fact and the other destination(Did1/Did2)

```

checkUser(Tuple, [H|T], Did, Sm, Cnt):-

```

```

string=(getUid(Tuple), getUid(H)), string=(Did,getDid(H)), R1 .=
getRate(Tuple), R2 .= getRate(H) , Dif .= -(R1,R2), X .=
cns2(+ (Sm,Dif), 1+(Cnt)) &X.

```

%recursive Case - to continue search

```

checkUser(Tuple, [H|T], Did, Sm, Cnt):& checkUser(Tuple, T, Did,
Sm, Cnt).

```

%base case

```

getSumCnt(Did1, Did2, [], Sm, Cnt):&cns2(Sm,Cnt).

%recursive Case - current fact matches destination 1 (Did1)
getSumCnt(Did1, Did2, [X/T], Sm, Cnt):-
=(check(X,Did1,Did2),1), [Sm1, Cnt1] . = checkUser(X, T, Did2, Sm,
Cnt) &getSumCnt(Did1, Did2, T, Sm1, Cnt1).

%recursive Case - current fact matches destination 2 (Did2)
getSumCnt(Did1, Did2, [X/T], Sm, Cnt):-
=(check(X,Did1,Did2),2), [Sm1, Cnt1] . = checkUser(X, T, Did1, Sm,
Cnt) &getSumCnt(Did1, Did2, T, Sm1, Cnt1).

%recursive Case - current fact does not match any of the
destinations
getSumCnt(Did1, Did2, [X/T], Sm, Cnt):-
=(check(X,Did1,Did2),0) &getSumCnt(Did1, Did2, T, Sm, Cnt).

```

Input/Output in Relfun

```

rfi-p> getSumCnt(dest01,
dest03,[[user1,dest03,4],[user0,dest03,4],[user2,dest03,4],[user
2,dest04,3],[user0,dest04,3],[user1,dest04,5],[user0,dest01,2],[
user1,dest01,2]] , 0, 0)
[4,2]

```

Applying this process to each pair in the list will produce another list as follows:

```
& cns([],[])
```

```
[Burma,Thimpu,2,2],[Thimpu,Delhi,-1,1],[Thimpu,Burma,-2,2],[Burma,Delhi,1,1]
```

The table below gives a visual representation of the deviation values that we will need to generate in order to predict a rating for Fred.

Destination		Sum	Count
Burma	Thimpu	$(4-3) + (2-1) = 2$	2
Thimpu	Delhi	$(3-4) = -1$	1

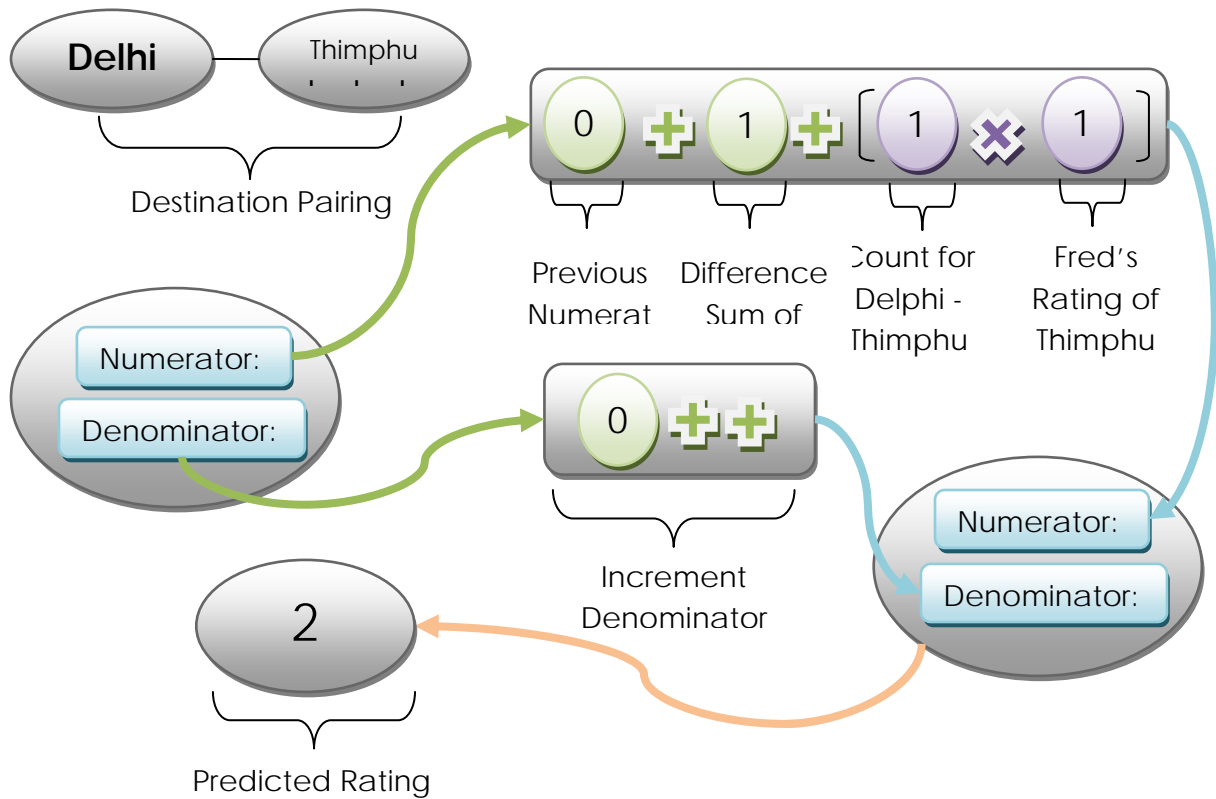
Thimpu	Burma	$(3-4) + (1-2) = -2$	2
Delhi	Thimpu	$(4-3) = 1$	1

Now that we have our deviation values, we can continue on to use these values to make a prediction. We'll also need to refer back to our original rating table to make this prediction.

	Burma	Tokyo	Thimpu	Delhi
Bob	4	-	3	4
Fred	2	-	1	?
Jack	-	3	-	5

As previously mentioned, we would like to predict how Fred would rate Delhi. To make this prediction we need to consider every pairing in our collection of deviation values where the first destination is Delhi.

Starting with a numerator and denominator of zero (as there is always at least one pair when these values are used there will never be a division by zero error), for each pairing we have, we increment the denominator, and add to the numerator the sum listed in the deviation table for that pair as well as the product of the count from the deviation table and the value that our user rated that item. After iterating through all common items, we return the quotient of the numerator and denominator as the predicted rating for that item.



In RELFUN this part should look somewhat like this:

```

predict(Destination,[Y|Deviations])
:- Y .=[Destination, Destination2, Sum, Count] , X . =
+(+(Numerator, Sum), *(Count, Rating)), Denominator . =
1+(Denominator)
& /(Numerator, Denominator).

```

As there was only one destination pairing beginning with Delhi in our set of deviation values, we now have our prediction of Fred's rating for Delhi. If we had more pairs, we would continue this process until all pairs beginning with Delhi were processed, incrementing the denominator each time.

Now, looking back at our original table, our value seems to be a very appropriate prediction.

	Burma	Tokyo	Thimpu	Delhi
Bob	4	–	3	4
Fred	2	–	1	2
Jack	–	3	–	5

Bob has a tendency to rate Destinations higher than Fred, but they seem to follow a similar pattern when making ratings, our prediction fits into this pattern.

References:

[1] Slope One Predictors for Online Rating-Based Collaborative Filtering-
Daniel Lemire

<http://www.daniel-lemire.com/fr/abstracts/SDM2005.html>

[2] inDiscover's Weighted Slope One Algorithm: Implementation Details and
Specification –

Sean McGrath

<http://www.indiscover.net/>

[3] Collaborative filtering made easy – *Slope One Algorithm Implemented in Python*
Bryan O'Sullivan

<http://www.serpentine.com/blog/2006/12/12/collaborative-filtering-made-easy/>

[4] RELFUN Project -

Harold Boley, et al.

<http://www.dfki.uni-kl.de/~vega/refun.html>