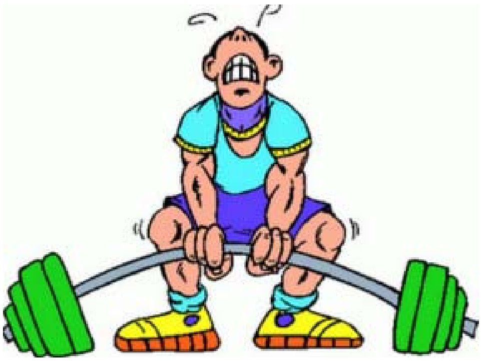
Master of Technology in Knowledge Engineering

Recommender Systems Workshop



Dr. Barry Shepherd
Institute of Systems Science
National University of Singapore
Email: barryshepherd@nus.edu.sg

Workshop Synopsis

We use the *movielens* database – this contains the ratings of 943 users on 1682 movies - the goal is to make movie recommendations to a test set of users.

1. Hand-code and test a simple ***user-based*** collaborative filtering recommender system using R (use my *starter* code if you wish)
2. Repeat using ***alternative similarity measures***, compare results
3. Convert your system to ***item-based*** collaborative filtering , compare results with step1 & 2

The MovieLens Data Set

- Each user has rated at least 20 movies
- Each ratings record has the format: UserID, MovieID, Rating, Timestamp
  - The data is randomly ordered. Users and items are numbered consecutively from 1.
  - Ratings are made on a 5-star scale (whole-star ratings only)
  - Timestamp is represented is seconds since 1/1/1970 UTC

UserID	movie	rating	datetime
1	61	4	878542420
1	189	3	888732928
1	33	4	878542699
1	160	4	875072547
1	20	4	887431883
1	202	5	875072442
1	171	5	889751711
1	265	4	878542441

1	Toy Story (1995)
2	GoldenEye (1995)
3	Four Rooms (1995)
4	Get Shorty (1995)
5	Copycat (1995)
6	Shanghai Triad (Yao a yao y
7	Twelve Monkeys (1995)
8	Babe (1995)
9	Dead Man Walking (1995)
10	Richard III (1995)

Get movie names from a separate file

MovieLens Dataset in Tabular Format

- I have converted the movie lens data set into tabular format*

User	movie1	movie2	movie3	movie4	movie5	movie6	etc….
1	2	5	4		3	1	
2		3		5	3	1	
3			2	3			

- Note that a much bigger dataset is also available containing 1,000,209 anonymous ratings of approximately 3,900 movies made by 6,040 MovieLens users who joined MovieLens in 2000.
  - Ambitious students may use this if you wish
  - This has not been converted to tabular format

  - ** Data is in the file: u_data_tabular.csv on IVLE*

The Basic User-User CF algorithm

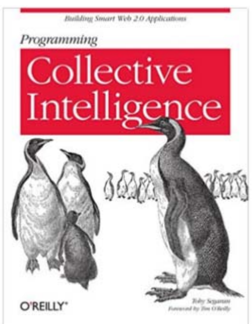
- Each user is represented by a single record (vector) containing a set of properties (features) – these typically the ratings or purchases of some of the items to be recommended (e.g. movies)
- To make a recommendation to a user
  - Compute the similarity of that user to all other users in the database (typically we use the Pearson coefficient)
  - For every item NOT rated or bought by the user
    - Compute the weighted average rating of all the other users for that item (or just consider the K nearest neighbours)
    - Weighted average = (Σ_{users} Item Rating * User Similarity)/ Σ_{users} User Similarity
  - Recommend the item with the biggest weighted average rating

A simple example using R

- First load the data* into a data frame

```
> users <- read.csv("simplemovies.csv")
> users
      user LadyInWater SnakesOnPlane JustMyLuck Superman Dupree NightListener
1     Rose         2.5           3.5        3.0      3.5    2.5           3.0
2  Seymour         3.0           3.5        1.5      5.0    3.5           3.0
3  Philips         2.5           3.0         NA      3.5     NA           4.0
4     Puig          NA           3.5        3.0      4.0    2.5           4.5
5  LaSalle         3.0           4.0        2.0      3.0    2.0           3.0
6 Matthews         3.0           4.0         NA      5.0    3.5           3.0
7     Toby          NA           4.5         NA      4.0    1.0            NA
>
```

*Example data taken from:
“Collective Intelligence”, O’Reilly



A simple example with sample R code

- To compute the similarity between users we use *cor(X,Y).* This computes the Pearson correlation between the columns of X and Y. Hence we first need to transpose the data using *t(..)* so that users are on the columns

```
> items <- as.data.frame(t(users[,2:ncol(users)]))
> colnames(items) <- users[,1]
> items
              Rose Seymour Philips Puig LaSalle Matthews Toby
LadyInWater    2.5     3.0     2.5   NA       3      3.0   NA
SnakesOnPlane  3.5     3.5     3.0  3.5       4      4.0  4.5
JustMyLuck     3.0     1.5      NA  3.0       2       NA   NA
Superman       3.5     5.0     3.5  4.0       3      5.0  4.0
Dupree         2.5     3.5      NA  2.5       2      3.5  1.0
NightListener  3.0     3.0     4.0  4.5       3      3.0   NA
>
```


A simple example with sample R code

- Set *use = "pairwise.complete.obs"* to ignore missing values during the correlation computation otherwise most correlations are *NA* (uncomputable)

```
> cor(items,items)
              Rose   Seymour Philips Puig   LaSalle Matthews Toby
Rose     1.0000000 0.3960590      NA   NA 0.5940885       NA   NA
Seymour  0.3960590 1.0000000      NA   NA 0.4117647       NA   NA
Philips         NA        NA      NA   NA        NA       NA   NA
Puig            NA        NA      NA   NA        NA       NA   NA
LaSalle  0.5940885 0.4117647      NA   NA 1.0000000       NA   NA
Matthews        NA        NA      NA   NA        NA       NA   NA
Toby            NA        NA      NA   NA        NA       NA   NA
>
> cor(items,items,use="pairwise.complete.obs")
              Rose   Seymour    Philips       Puig    LaSalle   Matthews       Toby
Rose     1.0000000 0.3960590  0.4045199 0.56694671  0.5940885 0.74701788  0.9912407
Seymour  0.3960590 1.0000000  0.2045983 0.31497039  0.4117647 0.96379568  0.3812464
Philips  0.4045199 0.2045983  1.0000000 1.00000000 -0.2581989 0.13483997 -1.0000000
Puig     0.5669467 0.3149704  1.0000000 1.00000000  0.5669467 0.02857143  0.8934051
LaSalle  0.5940885 0.4117647 -0.2581989 0.56694671  1.0000000 0.21128856  0.9244735
Matthews 0.7470179 0.9637957  0.1348400 0.02857143  0.2112886 1.00000000  0.6628490
Toby     0.9912407 0.3812464 -1.0000000 0.89340515  0.9244735 0.66284898  1.0000000
>
```

A simple example with sample R code

- Assembling into a function….

```
getrecommendations <- function(target) {

  # compute similarity between the target user and all other users
  sims <<- cor(items[,target],items[,!names(items) %in% c(target)],use="pairwise.complete.obs")
  # some users may have no ratings in common, this will cause NA's
  sims <<- sims[1,!is.na(sims)]
  # ignore users with similarity < 0 (avoid issues with the weighted mean below)
  sims <<- sims[sims >= 0]

  # for each item compute the weighted average of all the other user ratings
  wavrats = apply(items[,names(sims)],1,function(x) weighted.mean(x, sims, na.rm=TRUE))
  # some items may have no ratings at all, this will cause NA's
  wavrats = wavrats[!is.na(wavrats[])]

  # remove items already rated by the user
  notseenitems = row.names(items[is.na(items[,target]),])
  t = wavrats[notseenitems]

  sort(t[!is.na(t)] , decreasing = TRUE)[1:min(5,length(t))]  # get top 5 items
}
```

```
> getrecommendations("Toby")
NightListener    LadyInWater     JustMyLuck
     3.347790       2.832550       2.530981
>
```

The *apply()* function applies a function to every row (or column) in a data frame. Make sure you read the manual to understand fully how it works!

Testing the Recommendations (1)

- Split the available data into training and test sets
- Consider each test user in turn:

User	movie1	movie2	movie3	movie4	movie5	movie6	etc….
Test user	2	5	4		3	1	

For each movie rated by a test user:
- set the movie rating to blank (NA) – but keep a copy
- make a prediction for that movie using the training data
- compare the prediction with actual rating:
  error = abs(predicted rating – actual rating)
- keep a running total of the errors & number of tests:
  totalerror = totalerror + error
  cnt = cnt + 1
- restore the blank movie rating

Do this for all test users

At the end compute the overall MAE (mean absolute error)

MAE = totalerror/cnt

Example Testing Code

```
testusernames  = sample(names(items), 2) # identify 2 user randomly for testing
trainusernames = setdiff(names(items),testusernames) # take remaining users for training

#test recommendations for all users
testall <- function() {
  toterr = 0
  for (user in testusernames) {
    mae = testuser(user)
    cat("mae for ", user, "is ", mae, "\n");
    toterr = toterr + mae
  }
  cat(sprintf("AVERAGE MAE=%0.4f\n", toterr/length(testusernames)))
}

#test recommendations for one user
testuser <- function(target) {
  testitems  = row.names(items[!is.na(items[,target]),])
  targetdata  = items[testitems,target]
  names(targetdata) = testitems
  traindata = items[testitems,trainusernames]
  toterr = valid = 0
  for (item in testitems) {
    truerating = targetdata[item]
    targetdata[item] = NA
    sims = cor(targetdata,traindata,use="pairwise.complete.obs")
    sims = sims[,!is.na(sims)]
    prediction = weighted.mean(traindata[item,names(sims)], sims, na.rm=TRUE)
    if (!is.na(prediction)) {
      toterr = toterr + abs(prediction - unname(truerating))
      valid = valid + 1
    }
    targetdata[item] = truerating
  }
  return(toterr/valid)
}
```

To execute the testing code…

```
> testall()
mae for recommendations made to  Puig is  0.5556525
mae for recommendations made to  LaSalle is  0.4496651
AVERAGE MAE=0.5027
>
```


Testing the Recommendations (2)

- What does a MAE of (say) 1.19 mean in practice? Is it good or bad?
- We need to know how many predictions would actually be made and how many would likely be received favorably by the user?
- To answer this we need a Confusion Matrix!

Actual	Predictions	
	Won't Like	Will Like
Rated Poor	TN	FP
Rated High	FN	TP

KEY:
TN = true negative, FP = false positive
FN = false negative, TP = true positive

Row Sum = Total recommendations that could be made

Column Sum = Total recommendations that were made

Precision = TP/(TP+FP) Recall = TP/(TP+FN)

Deriving a Confusion Matrix

- Decide upon a rating threshold (T) to signify "likes"
  - E.g. A person likes a movie if they give it a rating >= 4
- Modify the test routine in order to keep 4 counts
  - TP (True Positive) ~ increment when predicted rating is >=T AND actual rating is >= T
  - FP (False Positive) ~ increment when predicted rating is >= T AND actual rating is < T
  - TN (True Negative) ~ increment when predicted rating is < T AND actual rating is < T
  - FN (False Negative) ~ increment when predicted rating is < T AND actual rating is >= T
- Increment the counts after each individual test (movie prediction) is made
- Display the counts as a confusion matrix at the end of the test

Coding the Confusion Matrix

```
#test recommendations for one user
testuserCM <- function(target,predthresh = 4, doprint=TRUE) {
  testitems   = row.names(items[!is.na(items[,target]),])
  targetdata  = items[testitems,target]
  names(targetdata) = testitems
  traindata = items[testitems,trainusernames]
  toterr = valid = TP = FP = TN = FN = 0
  for (item in testitems) {
    truerating = targetdata[item]
    targetdata[item] = NA
    sims = cor(targetdata,traindata,use="pairwise.complete.obs")
    sims = sims[,!is.na(sims)]
    prediction = weighted.mean(traindata[item,names(sims)], sims  , na.rm=TRUE)
    if (!is.na(prediction)) {
      toterr = toterr + abs(prediction - truerating)
      valid = valid + 1
      if (prediction >= predthresh) {
        if (truerating >= predthresh) TP = TP + 1
        else FP = FP + 1
      }
      else if (truerating >= predthresh) FN = FN + 1
      else TN = TN + 1
    }
    targetdata[item] = truerating
  }
  if (doprint) cat(" avgerr=",toterr/valid,"#preds=",valid,"\n")
  return(c(toterr/valid, TP, FP, TN, FN))
}
```

Add extra parameters

Set all counts to zero

Increment the TP,FP,TN,FN counts

One way to return multiple values is to use a vector

Coding the Confusion Matrix

```
#test recommendations for all users, with confusion matrix
testallCM <- function() {
  TP = FP = TN = FN = toterr = cnt = 0
  for (user in testusernames) {
    res = testuserCM(user, doprint=FALSE)
    toterr = toterr + res[1]
    TP = TP + res[2]
    FP = FP + res[3]
    TN = TN + res[4]
    FN = FN + res[5]
    cnt = cnt + 1
    cat(user, res,"\n")
  }
  cat(sprintf("MAE= %0.4f TP=%d FP=%d TN=%d FN=%d\n", toterr/cnt, TP, FP, TN, FN))
}
```

Add code to keep the global counts, and print the results

Basic Item-Item CF Algorithm

- Precompute the similarities between all items (use Euclidean distance)

```
> itemsims = apply(items, 1, function(item)
+     apply(items, 1, function(x) 1/(1+sqrt(sum((x - item)^2,na.rm=TRUE)))))
>
> itemsims
              LadyInWater SnakesOnPlane JustMyLuck  Superman    Dupree NightListener
LadyInWater     1.0000000     0.3483315  0.3483315 0.2402531 0.4494897     0.3874259
SnakesOnPlane   0.3483315     1.0000000  0.2553968 0.3090170 0.1886379     0.3203772
JustMyLuck      0.3483315     0.2553968  1.0000000 0.2079916 0.3203772     0.2989351
Superman        0.2402531     0.3090170  0.2079916 1.0000000 0.1918254     0.2526503
Dupree          0.4494897     0.1886379  0.3203772 0.1918254 1.0000000     0.2942981
NightListener   0.3874259     0.3203772  0.2989351 0.2526503 0.2942981     1.0000000
>
```

- The new getrecommendations() is:

```
getrecommendations2 <- function(username) {
  myRats = items[,username]
  wavrats = apply(itemsims, 1, function(simrow) weighted.mean(myRats, simrow, na.rm=TRUE))

  # remove items already rated by the user
  notseenitems = row.names(items[is.na(items[,username]),])
  t = wavrats[notseenitems]
  sort(t[!is.na(t)] , decreasing = TRUE)[1:min(5,length(t))]  # get top 5 items
}
```

```
> getrecommendations2("Toby")
NightListener    JustMyLuck   LadyInWater
     3.166743      2.936629      2.868767
>
```

** To get the same numbers as shown in the book and my slide p35 you must remove sqrt() from the similarity calculation (see book p11)*

Workshop Instructions (1)

1. Execute and test the sample *user-user CF* code on the movielens data
   - Randomly select 10 test users and 10 movies per user and compute the average MAE for their predicted ratings.
   - Increase the number of test users and test movies to obtain more accurate results. Keep a note of the time it takes to execute.
   - Repeat using Cosine Similarity and Euclidean distance similarity.
     (You will need to insert Euclidean distance code into the test code. Copy from the item-item CF code)
   - Which similarity measure (Pearson, Cosine, Euclidean) gives the least error?
     Which similarity measure is the fastest to execute?
2. Repeat above using the sample item-to-item CF code.
   - Do user-user and item-item CF give similar accuracy? Which is faster?
3. Modify the functions testall() and testuser() in order to derive a confusion matrix using a “like” threshold of 4; compute the precision and recall
   - What is the impact of changing the “*like*” threshold? (e.g. making it 3)
     (what is the best trade-off between precision and recall?)
4.

Workshop Instructions (2)

RecommenderLab is a very popular toolset for building recommendation engines using a variety of algorithms, e.g;

- collaborative filtering ~ user-user (UBCF), item-item (IBCF)
- matrix factorisation ~ ALS, SVD
- most popular items

<u>Workshop</u>

- Explore the recommenderLab sample code
  (you will need to install recommenderLab first)
- Do the results (e.g. average MAE) for movieLense data using User-User and Item-Item roughly agree with those obtained using my code on the previous pages?