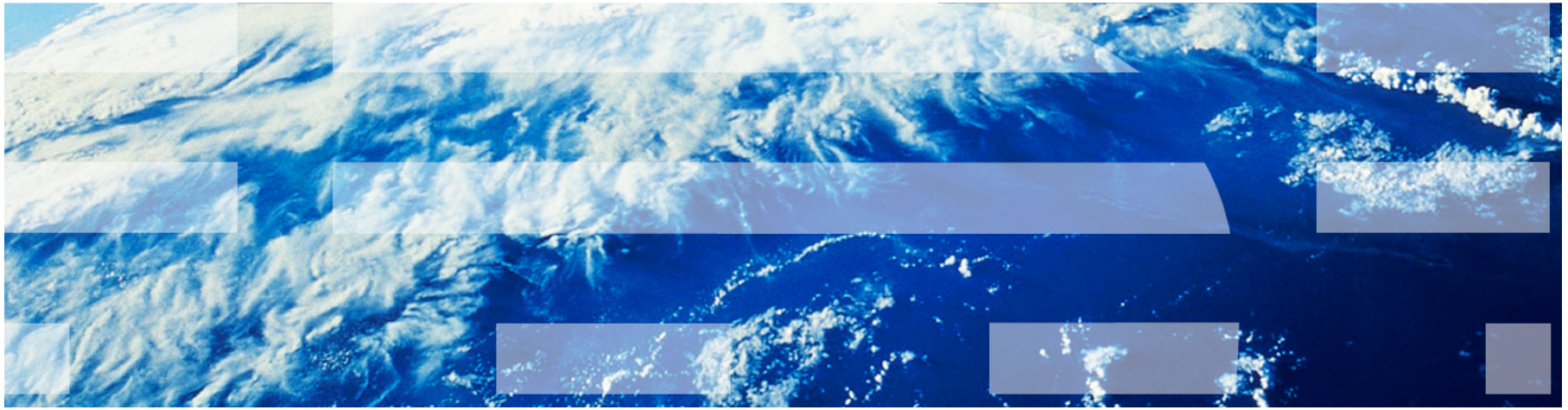
E6893 Big Data Analytics Lecture 4:

Big Data Analytics Algorithms -- I

Ching-Yung Lin, Ph.D.

Adjunct Professor, Dept. of Electrical Engineering and Computer Science

IBM Distinguished Researcher and Chief Scientist, Graph Computing



October 1st, 2015

Review — Key Components of Mahout

Collaborative Filtering

- User-Based Collaborative Filtering - single machine
- Item-Based Collaborative Filtering - **single machine** / MapReduce
- Matrix Factorization with Alternating Least Squares - **single machine** / MapReduce
- Matrix Factorization with Alternating Least Squares on Implicit Feedback- **single machine** / MapReduce
- Weighted Matrix Factorization, SVD++, Parallel SGD - **single machine**

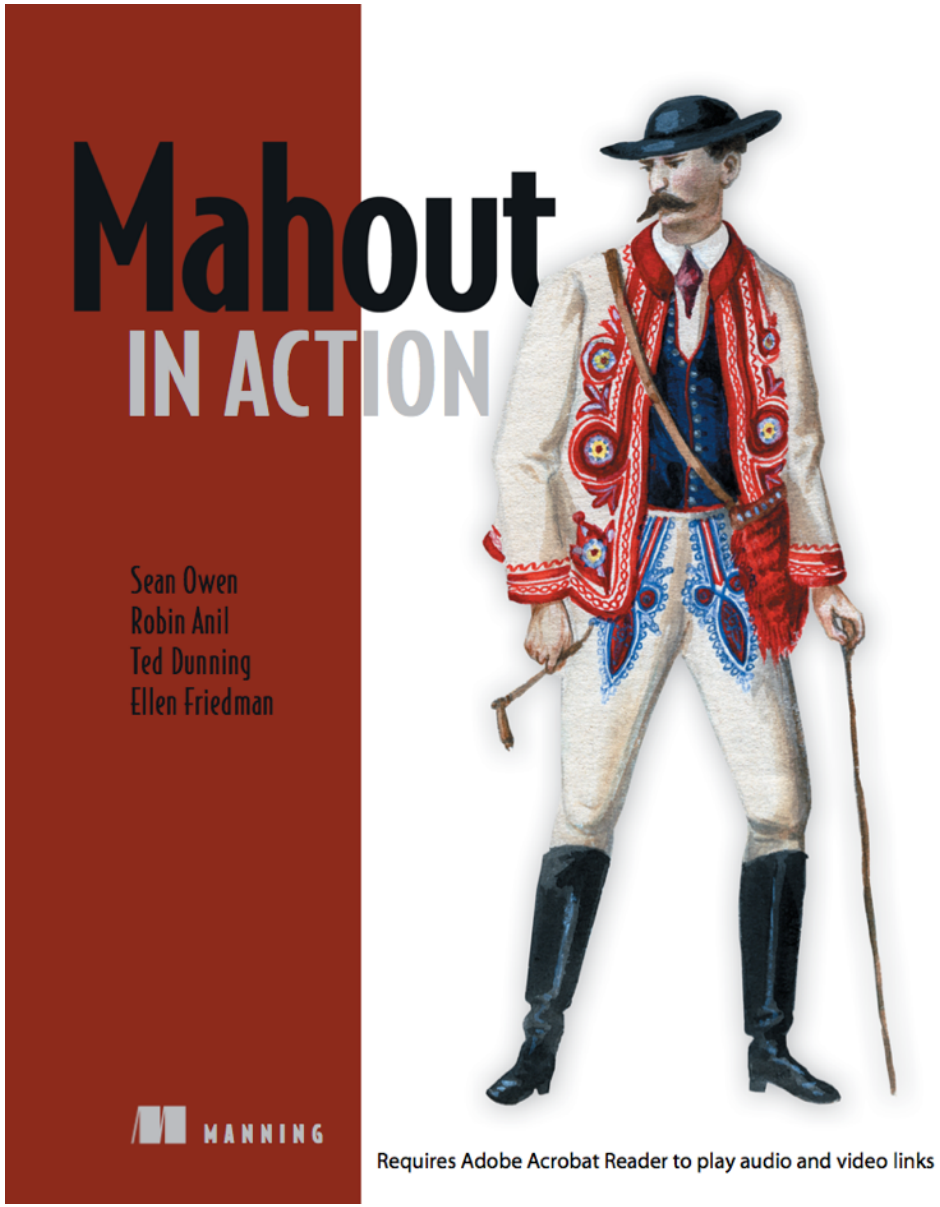
Classification

- Logistic Regression - trained via SGD - single machine
- Naive Bayes/ Complementary Naive Bayes - MapReduce
- Random Forest - **MapReduce**
- Hidden Markov Models - single machine
- Multilayer Perceptron - **single machine**

Clustering

- Canopy Clustering - **single machine** / MapReduce (deprecated, will be removed once Streaming k-Means is stable enough)
- k-Means Clustering - single machine / MapReduce
- Fuzzy k-Means - single machine / MapReduce
- Streaming k-Means - single machine / MapReduce
- Spectral Clustering - MapReduce

Mahout reference book



Requires Adobe Acrobat Reader to play audio and video links

Setting Up Mahout

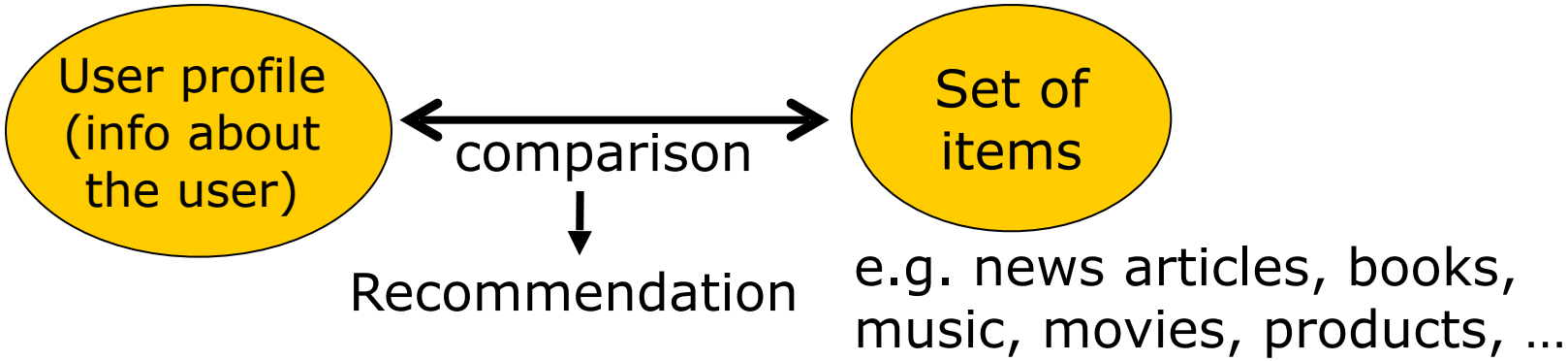
Step 1: Java JVM and IDEs (e.g., Eclipse)
Step 2: Maven
Step 3: Mahout

Eclipse Luna (June 2014)

Motivation – Information Overload

→ Provide users with the information they need at the right time

Recommender Systems: General Idea



Sample applications

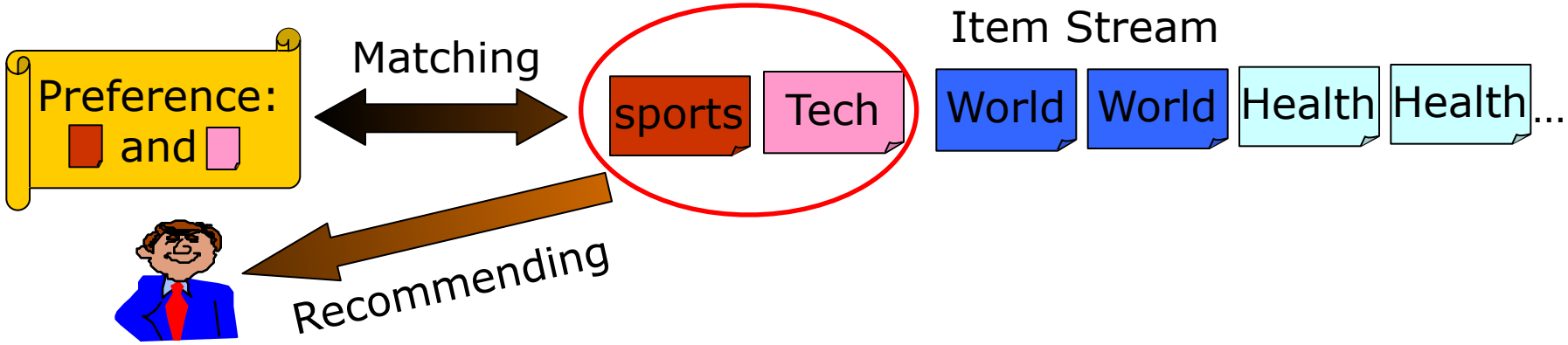
- E-commerce
  - Product recommender - Amazon
- Enterprise Activity Intelligence
  - Domain expert finder, …
- Digital Libraries
  - Pages/documents/books recommendation
- Personal Assistance
  - Museum guidance, …

Algorithms

- Content-based Filtering
- Collaborative Filtering
- Hybrid Filtering (combination of the above two)

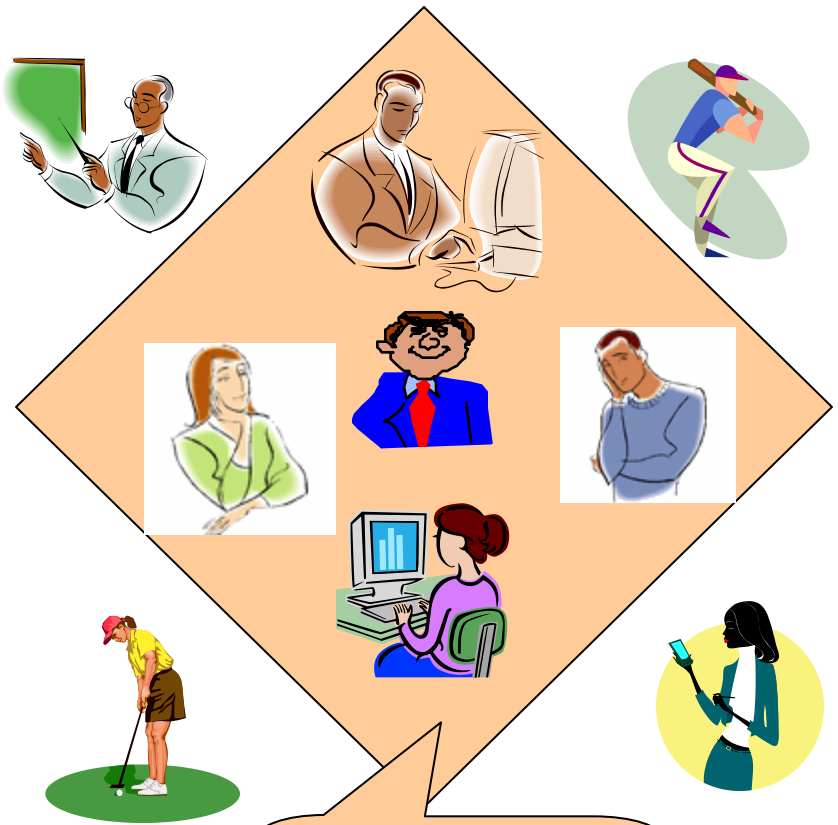
Content-based Filtering (CBF)

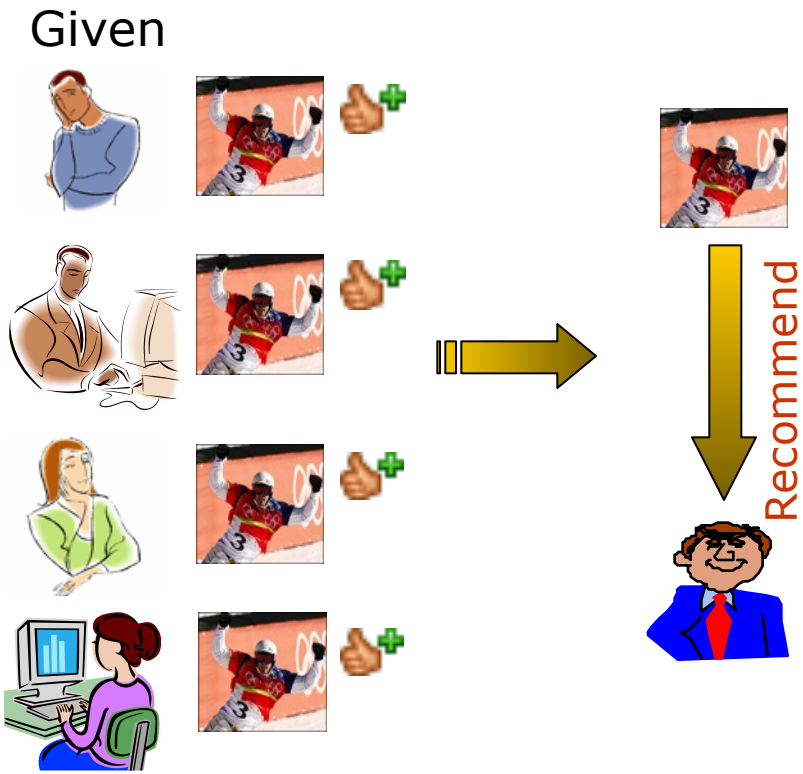
Recommending items based on the content and properties



Collaborative Filtering (CF)

Leveraging opinions of like-minded users





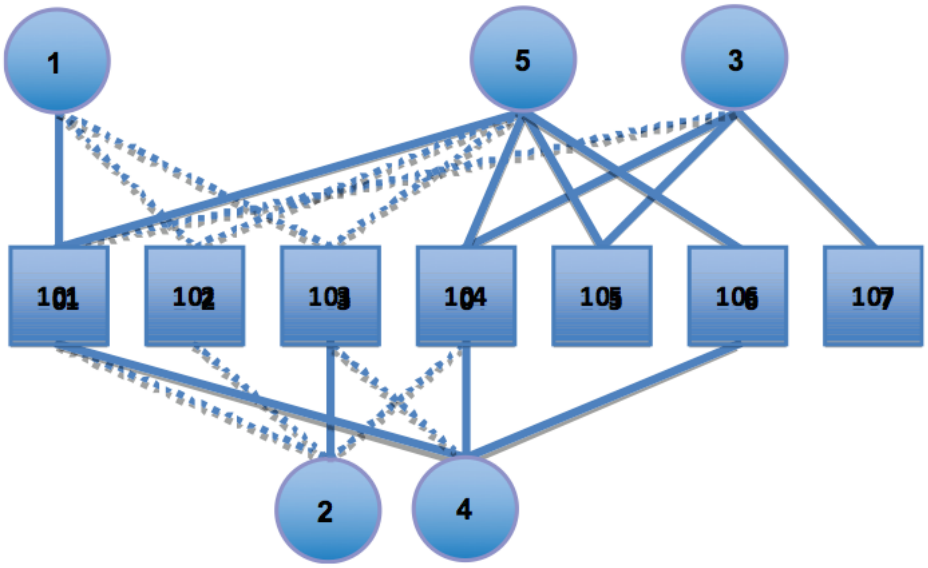
Exploiting Dynamic Patterns for Recommender Systems

- Dynamic nature from both items and users
  - Items expire over time with types of
    - Short-term
    - Long-term
  - Users' intentions of
    - Updating breaking information
    - Looking for long-term items
  - Users' interests evolve over time
- User adoption patterns
  - Some users are earlier adopters

Recommender — Inputs

```
1,101,5.0
1,102,3.0
1,103,2.5

2,101,2.0
2,102,2.5
2,103,5.0
2,104,2.0

3,101,2.5
3,104,4.0
3,105,4.5
3,107,5.0

4,101,5.0
4,103,3.0
4,104,4.5
4,106,4.0

5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0
```

Input Data: User, Item, Rating



Solid lines: positively related
Dashed lines: negatively related

User-based Recommendation — Scenario I

ADULT: *I'm looking for a CD for a teenager.*

EMPLOYEE: *OK, what does this teenager like?*

ADULT: *Oh, you know, what all the young kids like these days.*

EMPLOYEE: *What kind of music or bands?*

ADULT: *It's all noise to me. I don't know.*

EMPLOYEE: *Uh, well... I guess lots of young people are buying this boy band album here by New 2 Town?*

ADULT: *Sold!*

gettofail.com

User-based Recommendation — Scenario II

ADULT: *I'm looking for a CD for a teenage boy.*

EMPLOYEE: *What kind of music or bands does he like?*

ADULT: *I don't know, but his best friend is always wearing a Bowling In Hades T-shirt.*

EMPLOYEE: *Ah yes, a very popular nu-metal band from Cleveland. Well, we do have the new Bowling In Hades best-of album over here,* Impossible Split: The Singles 1997–2000...

User-based Recommendation — Scenario III

ADULT: *I'm looking for a CD for a teenage boy.*

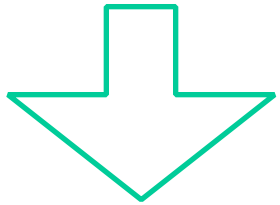
EMPLOYEE: *What kind of music or bands does he like?*

ADULT: *"Music?" Ha, well, I wrote down the bands from posters on his bedroom wall. The Skulks, Rock Mobster, the Wild Scallions… mean anything to you?*

EMPLOYEE: *I see. Well, my kid is into some of those albums too. And he won't stop talking about some new album from Diabolical Florist, so maybe…*

User-based Recommendation Algorithms

```
for every item i that u has no preference for yet
  for every other user v that has a preference for i
    compute a similarity s between u and v
    incorporate v's preference for i, weighted by s, into a running average
return the top items, ranked by weighted average
```



```
for every other user w
  compute a similarity s between u and w
  retain the top users, ranked by similarity, as a neighborhood n
for every item i that some user in n has a preference for,
      but that u has no preference for yet
  for every other user v in n that has a preference for i
    compute a similarity s between u and v
    incorporate v's preference for i, weighted by s, into a running average
```

Example Recommender Code via Mahout

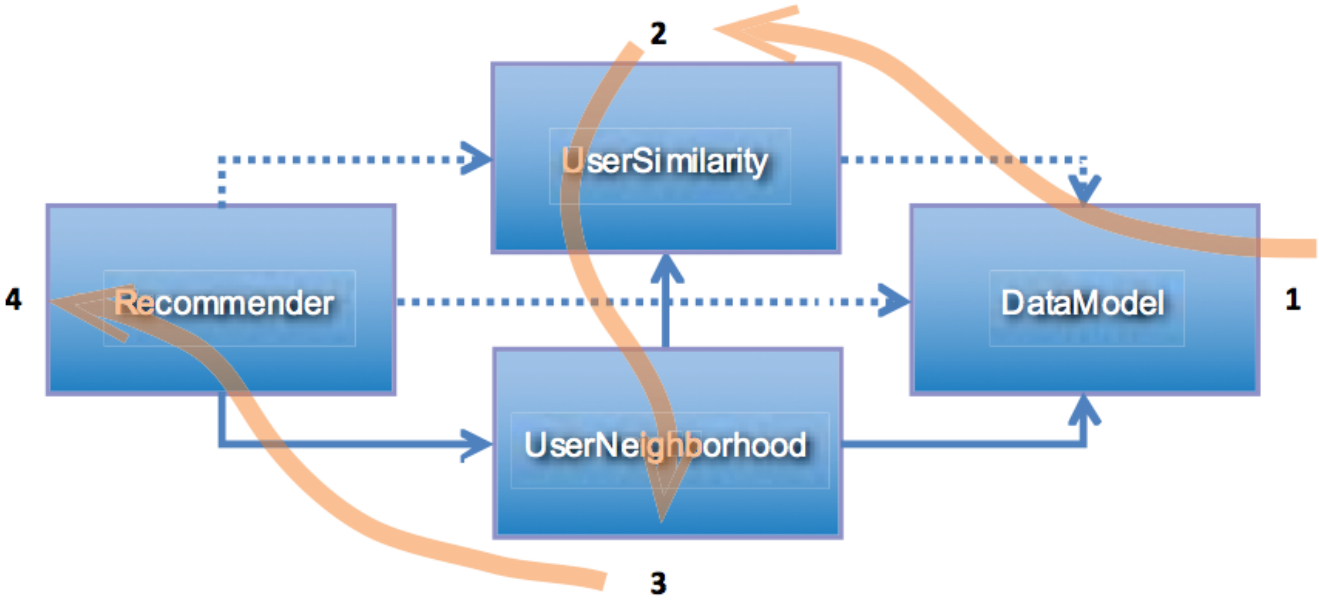
```
class RecommenderIntro {

  public static void main(String[] args) throws Exception {

    DataModel model =
      new FileDataModel (new File("intro.csv"));

    UserSimilarity similarity =
      new PearsonCorrelationSimilarity (model);
    UserNeighborhood neighborhood =
       new NearestNUserNeighborhood (2, similarity, model);

     Recommender recommender = new GenericUserBasedRecommender (
         model, neighborhood, similarity);

     List<RecommendedItem> recommendations =
         recommender.recommend(1, 1);

     for (RecommendedItem recommendation : recommendations) {
       System.out.println(recommendation);
     }

   }

}
```

Load data file

Create recommender engine

For user 1, recommend 1 item

Process and output of the example




```
RecommendedItem [item:104, value:4.257081]
```

Recommendation for Person 1:
Item 104 > Item 106
Item 107 is not favored

Refresh (Reload) Data

```
DataModel dataModel = new FileDataModel(new File("input.csv");
Recommender recommender = new SlopeOneRecommender(dataModel);
...
recommender.refresh(null);
```

Refresh DataModel, then itself

Note that `FileDataModel` will only reload data from the underlying file when asked to do so. It won't automatically detect updates or regularly attempt to reload the file's contents, for performance reasons. This is what the `refresh()` method is for. You probably don't want to cause only the `FileDataModel` to refresh, but also any objects that depend on its data. That's why, in practice, you'll almost surely call `refresh()` on a `Recommender` as follows.

Update data

`FileDataModel` supports update files. These are just more data files that are read after the main data file, and that overwrite any previously read data. New preferences are added and existing ones are updated. Deletes are handled by providing an empty preference value string.

For example, consider the following update file:

```
1,108,3.0
1,103,
```

This says, "update (or create) user 1's preference for item 108, and set the value to 3.0" and "remove user 1's preference for item 103."

These update files must simply exist in the same directory as the main data file, and their names must begin with the same prefix, up to the first period. For example, if the main data file is foo.txt.gz, the update files might be named foo.1.txt.gz and foo.2.txt.gz.

User Similarity Measurements

- Pearson Correlation Similarity
- Euclidean Distance Similarity
- Cosine Measure Similarity
- Spearman Correlation Similarity
- Tanimoto Coefficient Similarity (Jaccard coefficient)
- Log-Likelihood Similarity

Pearson Correlation Similarity

$$\rho_{X,Y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y}$$

Data:

```
1,101,5.0
1,102,3.0
1,103,2.5

2,101,2.0
2,102,2.5
2,103,5.0
2,104,2.0

3,101,2.5
3,104,4.0
3,105,4.5
3,107,5.0

4,101,5.0
4,103,3.0
4,104,4.5
4,106,4.0

5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0
```

	Item 101	Item 102	Item 103	Correlation with user 1
User 1	5.0	3.0	2.5	1.000
User 2	2.0	2.5	5.0	-0.764
User 3	2.5	-	-	-
User 4	5.0	-	3.0	1.000
User 5	4.0	3.0	2.0	0.945

missing data

On Pearson Similarity

Three problems with the Pearson Similarity:

1. Not take into account of the number of items in which two users' preferences overlap. (e.g., 2 overlap items ==> 1, more items may not be better.)
2. If two users overlap on only one item, no correlation can be computed.
3. The correlation is undefined if either series of preference values are identical.

Adding Weighting.WEIGHTED as 2nd parameter of the constructor can cause the resulting correlation to be pushed towards 1.0, or -1.0, depending on how many points are used.

Euclidean Distance Similarity

$$\mathrm{d}(\mathbf{p},\mathbf{q}) = \mathrm{d}(\mathbf{q},\mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2} = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}.$$

Similarity = 1 / (1 + d)

	Item 101	Item 102	Item 103	Distance	Similarity to user 1
User 1	5.0	3.0	2.5	0.000	1.000
User 2	2.0	2.5	5.0	3.937	0.203
User 3	2.5	-	-	2.500	0.286
User 4	5.0	-	3.0	0.500	0.667
User 5	4.0	3.0	2.0	1.118	0.472

Cosine Similarity

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$$

Cosine similarity and Pearson similarity get the same results if data are normalized (mean == 0).

Spearman Correlation Similarity

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}.$$

$$d_i = x_i - y_i$$

Pearson value on the relative ranks

Example for ties

Variable X_i	Position in the ascending order	Rank x_i
0.8	1	1
1.2	2	$\frac{2+3}{2} = 2.5$
1.2	3	$\frac{2+3}{2} = 2.5$
2.3	4	4
18	5	5

	Item 101	Item 102	Item 103	Correlation to user 1
User 1	3.0	2.0	1.0	1.0
User 2	1.0	2.0	3.0	–1.0
User 3	1.0	-	-	-
User 4	2.0	-	1.0	1.0
User 5	3.0	2.0	1.0	1.0

Caching User Similarity

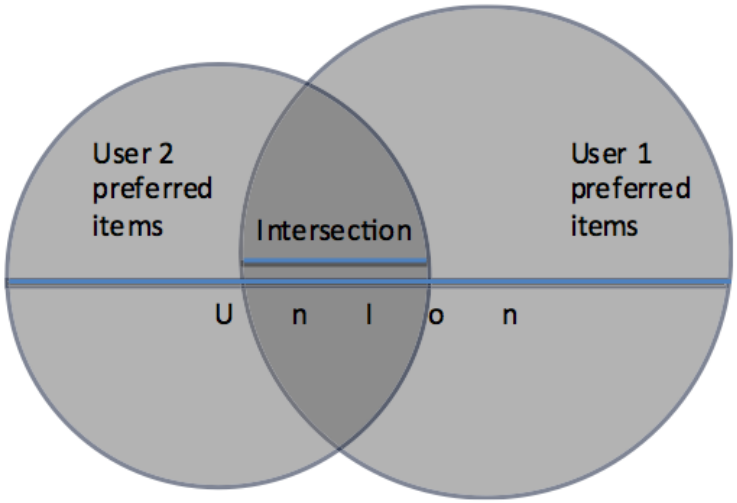
```
UserSimilarity similarity = new CachingUserSimilarity(
    new SpearmanCorrelationSimilarity(model), model);
```

Spearman Correlation Similarity is time consuming.
Need to use Caching ==> remember s user-user similarity which was previously computed.

Tanimoto (Jaccard) Coefficient Similarity

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Discard preference values



	Item 101	Item 102	Item 103	Item 104	Item 105	Item 106	Item 107	Similarity to user 1
User 1	X	X	X					1.0
User 2	X	X	X	X				0.75
User 3	X			X	X		X	0.17
User 4	X		X	X		X		0.4
User 5	X	X	X	X	X	X		0.5

Tanimoto similarity is the same as Jaccard similarity. But, Tanimoto distance is not the same as Jaccard distance.

Log-Likelihood Similarity

Asses how unlikely it is that the overlap between the two users is just due to chance.

$$D = -2 \ln \left(\frac{\text{likelihood for null model}}{\text{likelihood for alternative model}} \right)$$
$$= -2 \ln(\text{likelihood for null model}) + 2 \ln(\text{likelihood for alternative model})$$

	Item 101	Item 102	Item 103	Item 104	Item 105	Item 106	Item 107	Similarity to user 1
User 1	X	X	X					0.90
User 2	X	X	X	X				0.84
User 3	X			X	X		X	0.55
User 4	X		X	X		X		0.16
User 5	X	X	X	X	X	X		0.55

Performance measurements

Using GroupLens data (http://grouplens.org): 10 million rating MovieLens dataset.

- Spearnman: 0.8
- Tanimoto: 0.82
- Log-Likelihood: 0.73
- Euclidean: 0.75
- Pearson (weighted): 0.77
- Pearson: 0.89

	Item 1	Item 2	Item 3
Actual	3.0	5.0	4.0
Estimate	3.5	2.0	5.0
Difference	0.5	3.0	1.0
Average difference	$= (0.5 + 3.0 + 1.0) / 3 = 1.5$		
Root-mean-square	$= \sqrt{((0.5^2 + 3.0^2 + 1.0^2) / 3)} = 1.8484$		

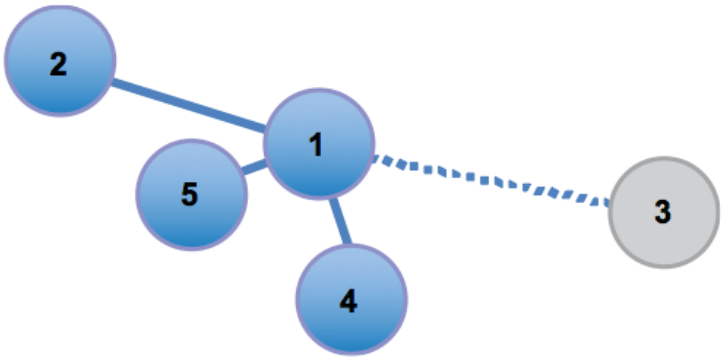
Performance measurements

```
DataModel model = new GroupLensDataModel (new File("ratings.dat"));
RecommenderEvaluator evaluator =
  new AverageAbsoluteDifferenceRecommenderEvaluator ();
RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
  @Override
  public Recommender buildRecommender(
        DataModel model) throws TasteException {
    UserSimilarity similarity = new PearsonCorrelationSimilarity(model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood(100, similarity, model);
    return new GenericUserBasedRecommender(
        model, neighborhood, similarity);
  }
};
double score = evaluator.evaluate(
    recommenderBuilder, null, model, 0.95, 0.05);
System.out.println(score);
```

10 nearest neighbors: 0.98
100 nearest neighbors: 0.89
500 nearest neighbors: 0.75

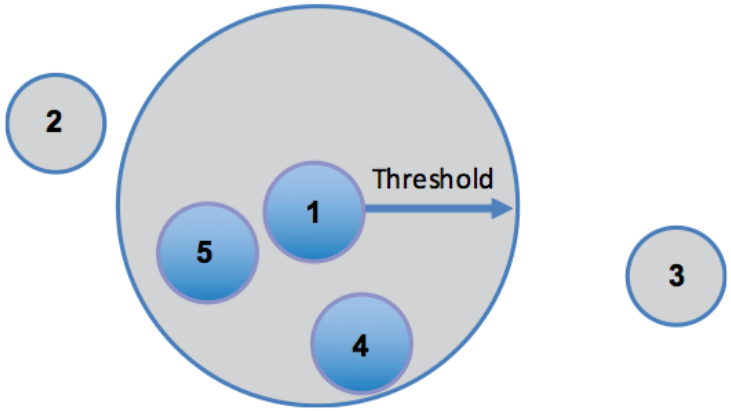
95% of training; 5% of testing

Selecting the number of neighbors



Based on number of neighbors

```
new NearestNUserNeighborhood(100, similarity, model);
```



Based on a fixed threshold, e.g., 0.7 or **0.5**

```
new ThresholdUserNeighborhood(0.7, similarity, model)
```

Item-based recommendation

ADULT: *I'm looking for a CD for a teenage boy.*

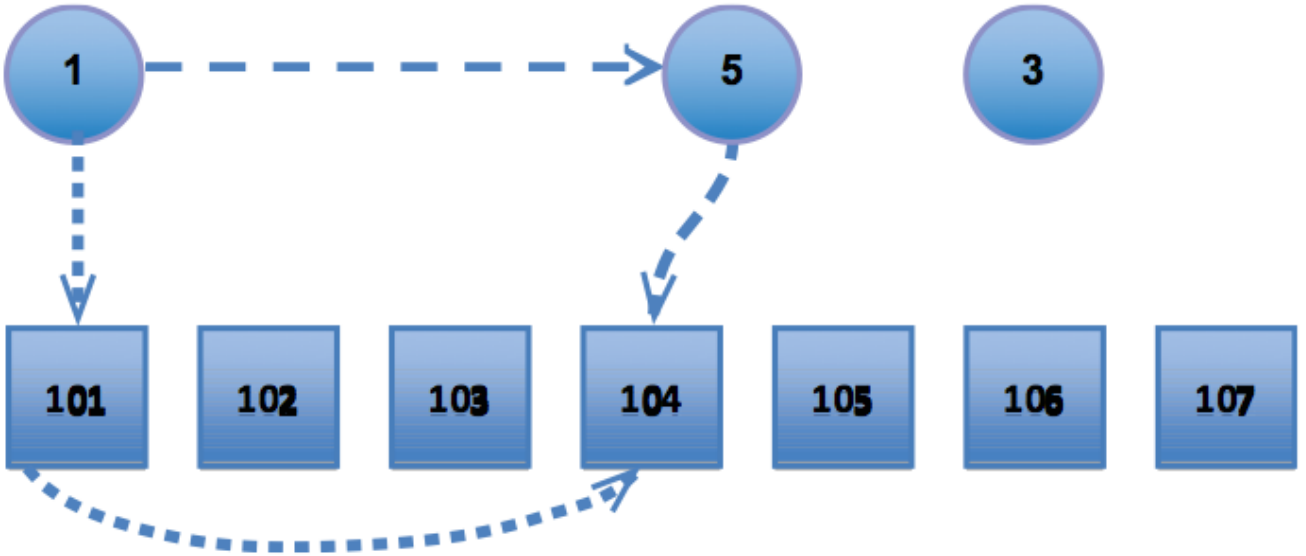
EMPLOYEE: *What kind of music or bands does he like?*

ADULT: *He wears a Bowling In Hades T-shirt all the time and seems to have all of their albums. Anything else you'd recommend?*

EMPLOYEE: *Well, about everyone I know that likes Bowling In Hades seems to like the new Rock Mobster album.*

Item-based recommendation algorithm

```
for every item i that u has no preference for yet
  for every item j that u has a preference for
    compute a similarity s between i and j
    add u's preference for j, weighted by s, to a running average
return the top items, ranked by weighted average
```



Code and Performance of Item-Based Recommendation

```
public Recommender buildRecommender(DataModel model)
    throws TasteException {
  ItemSimilarity similarity = new PearsonCorrelationSimilarity(model);
  return new GenericItemBasedRecommender(model, similarity);
}
```

performance

Implementation	Similarity
PearsonCorrelationSimilarity	0.75
PearsonCorrelationSimilarity + weighting	0.75
EuclideanDistanceSimilarity	0.76
EuclideanDistanceSimilarity + weighting	0.78
TanimotoCoefficientSimilarity	0.77
LogLikelihoodSimilarity	0.77

One thing you may notice is that this recommender setup runs significantly faster. That's not surprising, given that the data set has about 70,000 users and 10,000 items. Item-based recommenders are generally faster when there are fewer items than users.

Slope-One Recommender

For example, let's say that, on average, people rate *Scarface* higher by 1.0 than *Carlito's Way*. Let's also say that everyone rates *Scarface* the same as *The Godfather*, on average. And now, there's a user who rates *Carlito's Way* as 2.0, and *The Godfather* as 4.0. What's a good estimate of their preference for *Scarface*?

Based on *Carlito's Way*, a good guess would be 2.0 + 1.0 = 3.0. Based on *The Godfather*, it might be 4.0 + 0.0 = 4.0. A better guess still might be the average of the two: 3.5. This is the essence of the slope-one recommender approach.

the assumption that there's some linear relationship between the preference values for one item and another, that it's valid to estimate the preferences for some item Y based on the preferences for item X, via some linear function like $Y = mX + b$. Then the slope-one recommender makes the additional simplifying assumption that m=1: slope one.

Slope-One Algorithm

```
for every item i
  for every other item j
    for every user u expressing preference for both i and j
      add the difference in u's preference for i and j to an average

for every item i the user u expresses no preference for
  for every item j that user u expresses a preference for
    find the average preference difference between j and i
    add this diff to u's preference value for j
    add this to a running average
return the top items, ranked by these averages
```

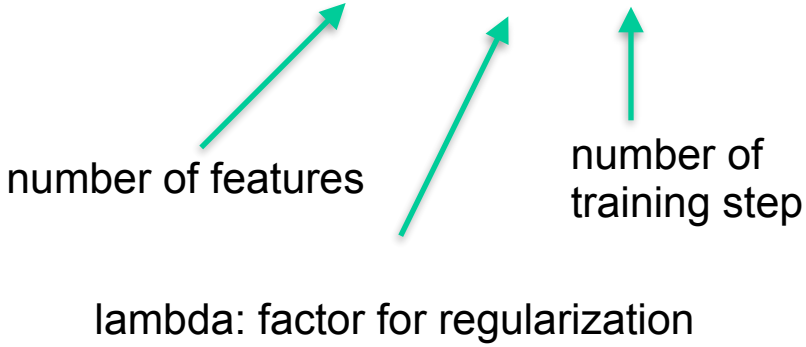
Difference values from the example

	Item 101	Item 102	Item 103	Item 104	Item 105	Item 106	Item 107
Item 101		–0.833	0.875	0.25	0.75	–0.5	2.5
Item 102			0.333	0.25	0.5	1.0	-
Item 103				0.167	1.5	1.5	-
Item 104					0.0	–0.25	1.0
Item 105						0.5	0.5
Item 106							-
Item 107							

Slope-One got a result of near 0.65 on the GroupLens data

Other recommenders — SVD recommender

```
new SVDRecommender(model, new ALSWRFactorizer(model, 10, 0.05, 10))
```



SVD method got 0.69 on the GroupLens data

Linear Interpolation Item-based recommender

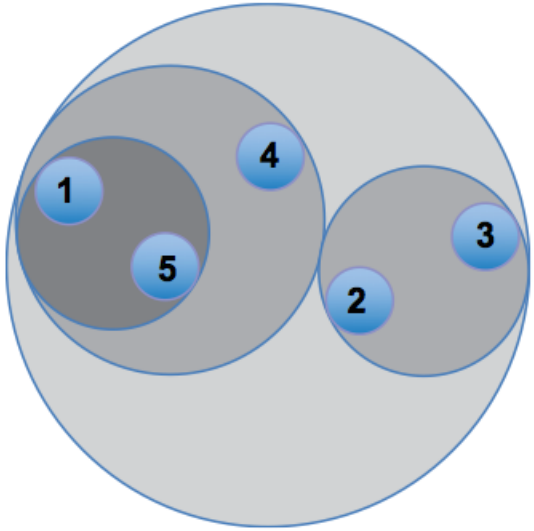
`KnnItemBasedRecommender` still estimates preference values by means of a weighted average of the items the user already has a preference for, but the weights aren't the results of a similarity metric. Instead, the algorithm calculates the optimal set of weights to use between all pairs of items by means of some linear algebra

```
ItemSimilarity similarity = new LogLikelihoodSimilarity(model);
Optimizer optimizer = new NonNegativeQuadraticOptimizer();
return new KnnItemBasedRecommender(model, similarity, optimizer, 10);
```

SVD method got 0.76 on the GroupLens data

Cluster-based Recommendation

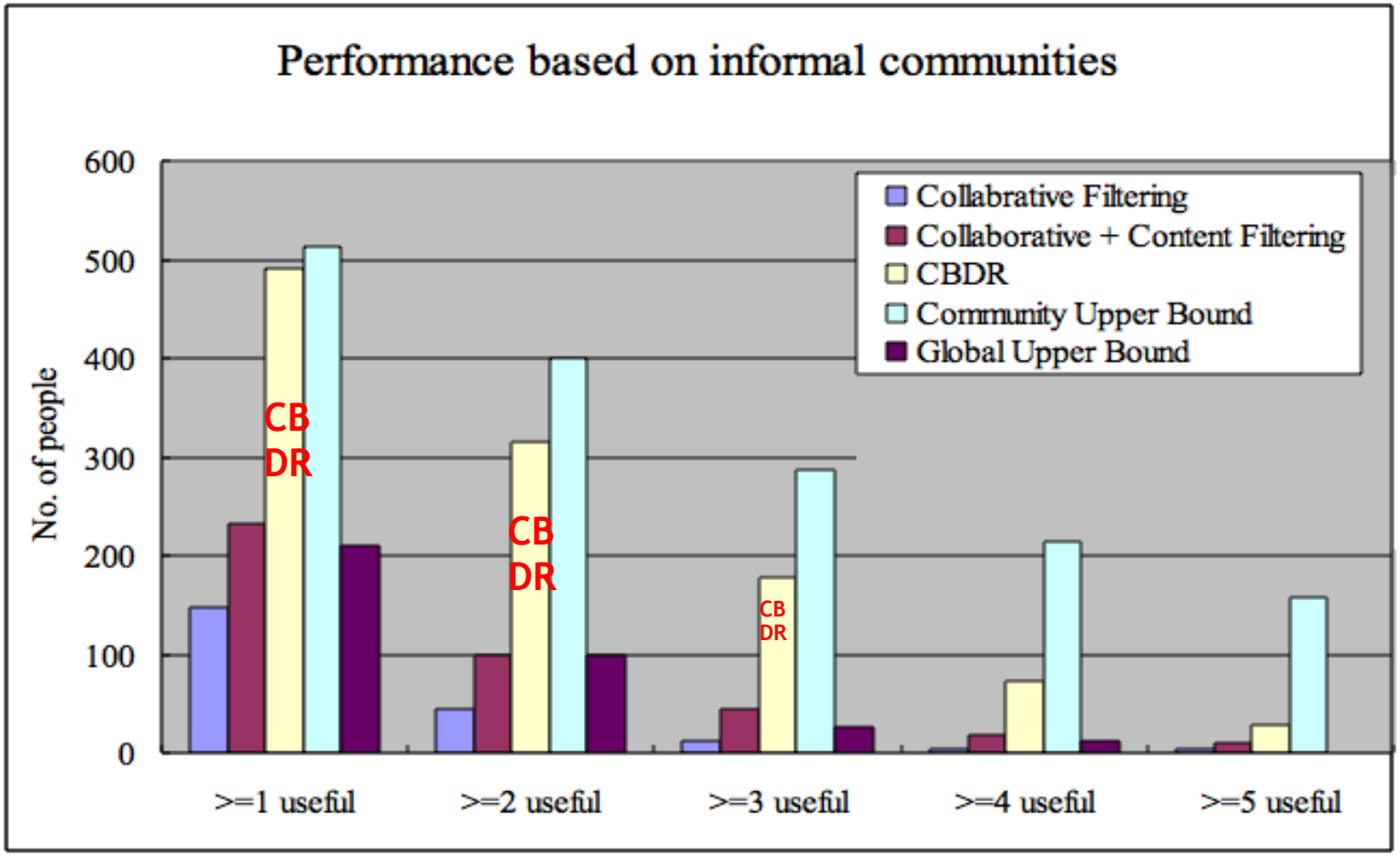
```
UserSimilarity similarity = new LogLikelihoodSimilarity(model);
ClusterSimilarity clusterSimilarity =
    new FarthestNeighborClusterSimilarity(similarity);
return new TreeClusteringRecommender(model, clusterSimilarity, 10);
```



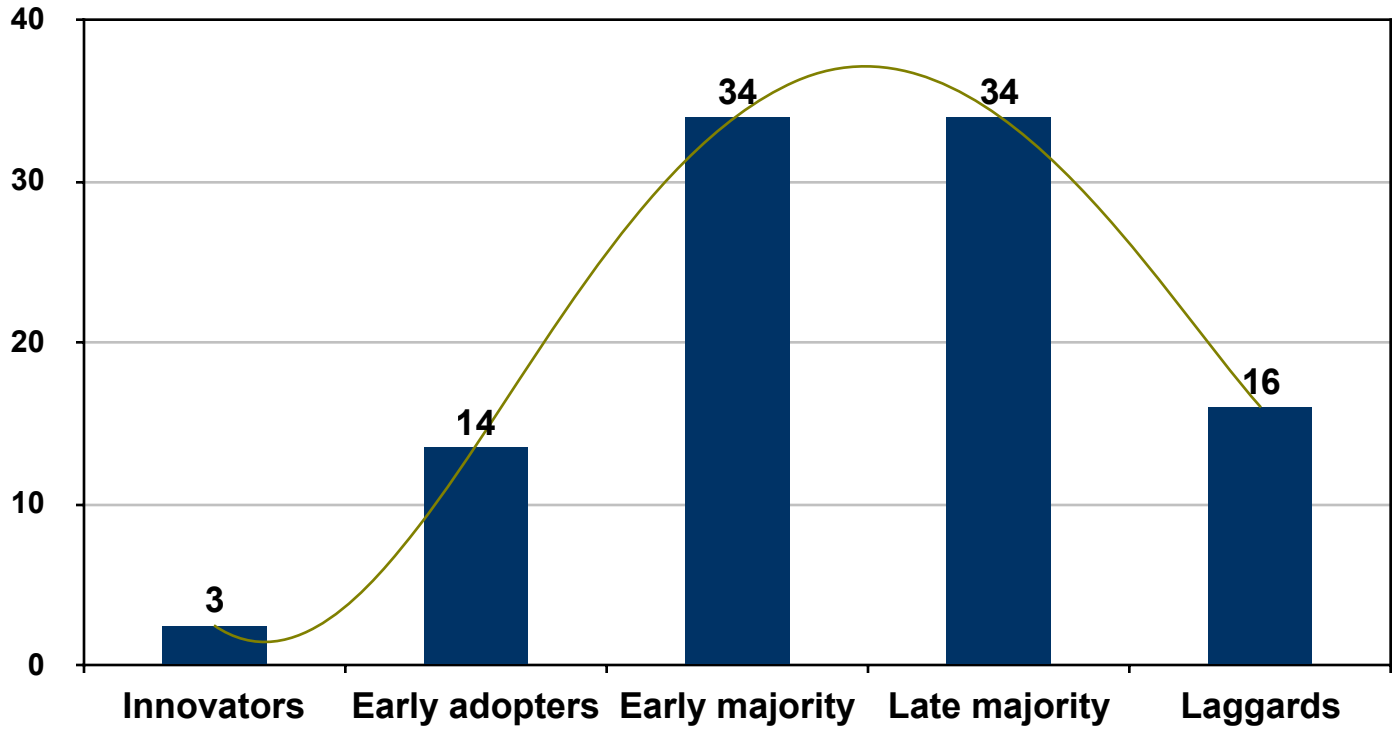
Other Recommenders not in Mahout — Groups (SDM 06)

- A 3rd party Knowledge Repository: 30K users and 20K documents.
Study the most active 697 users who have at least 20 download in a year.

- **Results: beyond Collaborative Filtering: (1) Collaborative + Content Filtering (53% improvement); (2) CBDR: Collaborative + Content Filtering + Graph Community Analytics (259% accuracy improvement over collaborative filtering)**



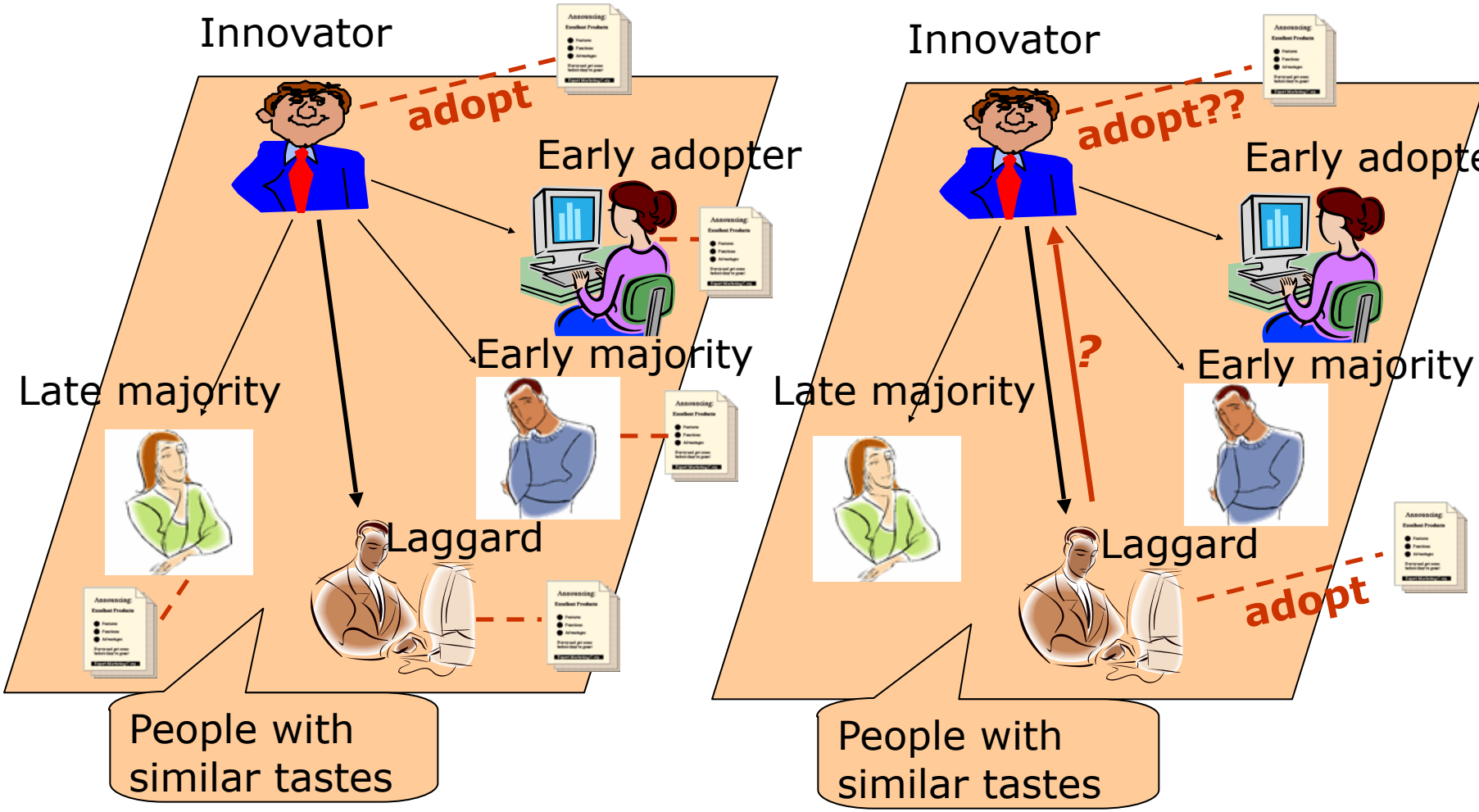
In E-Commerce: Rogers' Diffusion of Innovations Theory



Users' adoption patterns: Some users tend to adopt innovations earlier than others

→ Information virtually flows from early adopters to late adopters

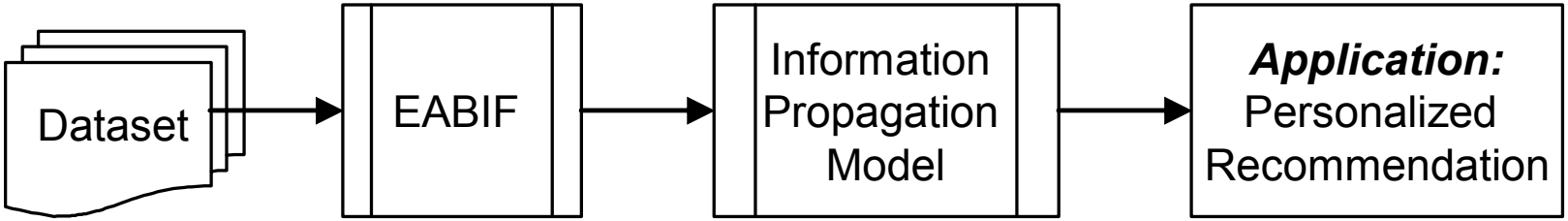
Recommendation Driven by Information Flow



Influence is not symmetric!

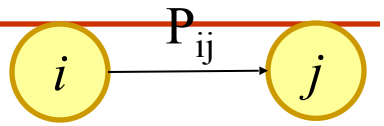
Scheme Overview

- Leverage the uneven influence



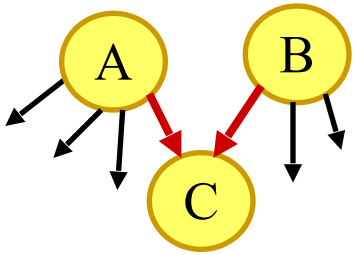
EABIF: Early Adoption based Information Flow Network

EABIF (1) -- Markov Chain



- A Markov chain has two components
  - A network structure where each node is called a state
  - A transition probability of traversing a link given that the chain is in a state (P: transition probability matrix)
- A stationary distribution is a probability distribution q such that q = qP → how likely you will stay at one node
- Application -- PageRank [Brin and Page '98]
  - Assumption: A link from page A to page B is a recommendation of page B by the author of A

    → Quality of a page is related to
    - Number of pages linking to it
    - The quality of pages linking to it
  - Assume the web is a Markov chain
    - PageRank = stationary distribution of this Markov chain



EABIF (2)

- Early Adoption Matrix (EAB)
  - Count how many items one user accesses earlier than the other – pairwise comparison
- Markov Chain Model
  - Normalize EAB to a transition matrix F of a Markov chain
  - Adjustment F to guarantee the existence of stationary distribution of the Markov chain
    - Make the matrix stochastic $\sum_j \bar{F}_{ij} = 1$
    - Make the Markov chain irreducible $\bar{\bar{F}}_{ij} \neq 0, \ 1 \leq i, j \leq N$

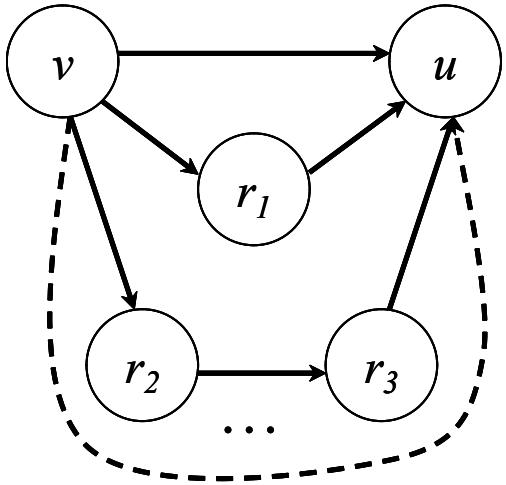
Information Propagation Models

1. Summation of various propagation steps

$$\mathbf{F}_{if(m)} = \left(\bar{\bar{\mathbf{F}}} + \bar{\bar{\mathbf{F}}}^{(2)} + \cdots + \bar{\bar{\mathbf{F}}}^{(m)}\right) / m$$

2. Direct summation

$$\mathbf{F}_{if(d)} = \frac{\bar{\bar{\mathbf{F}}} \cdot \left(\mathbf{I} - \bar{\bar{\mathbf{F}}}^{(N-1)}\right)}{\mathbf{I} - \bar{\bar{\mathbf{F}}}}$$



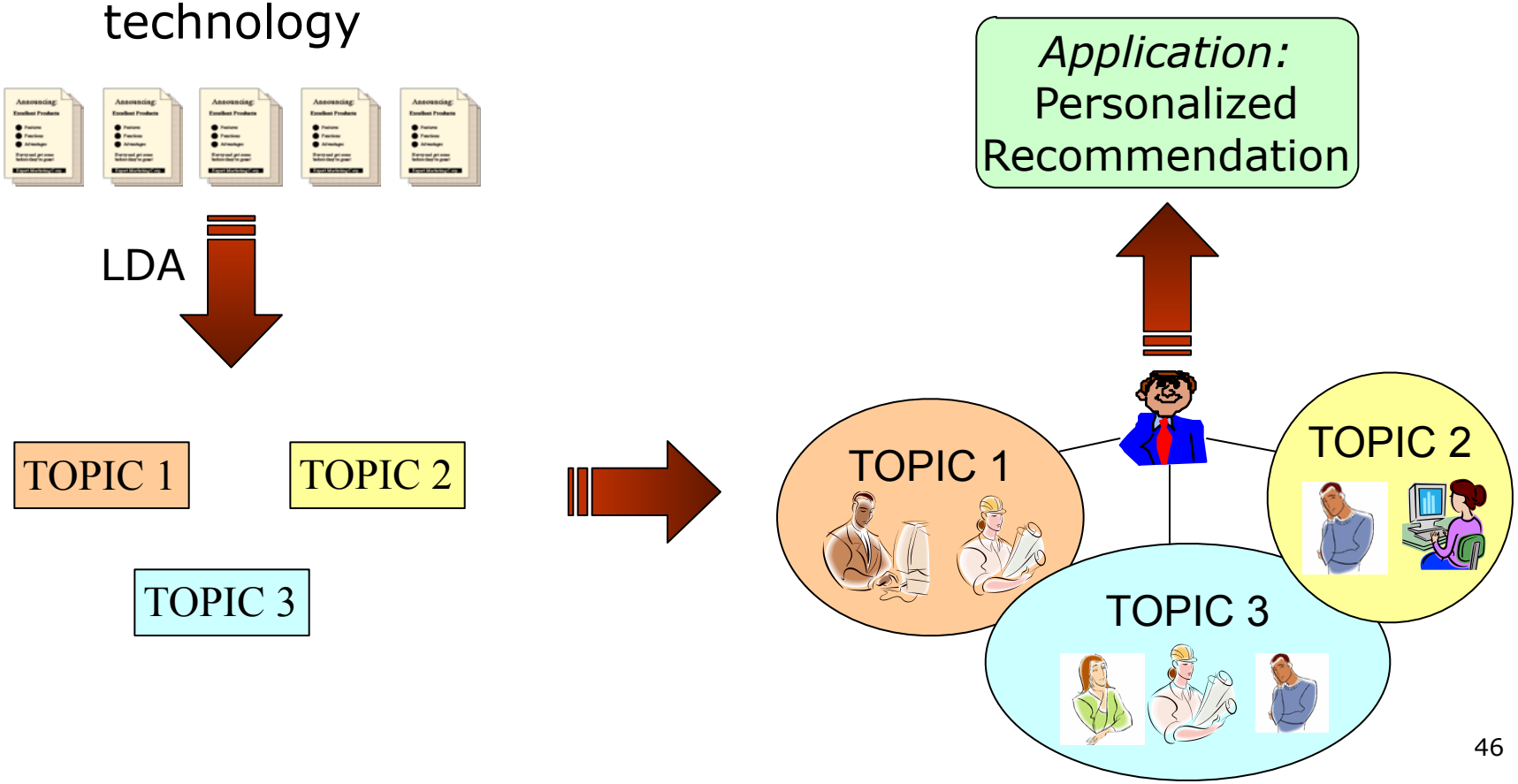
N: number of the nodes

3. Exponential weighted summation

$$\mathbf{F}_{if(\exp)} = \left(\left(\beta\bar{\bar{\mathbf{F}}}\right) + \frac{1}{2!}\left(\beta\bar{\bar{\mathbf{F}}}\right)^2 + \cdots + \frac{1}{(N-1)!}\left(\beta\bar{\bar{\mathbf{F}}}\right)^{(N-1)} + \cdots\right) \cdot \exp(-\beta)$$

Topic-Sensitive Early Adoption Based Information Flow (TEABIF) Network

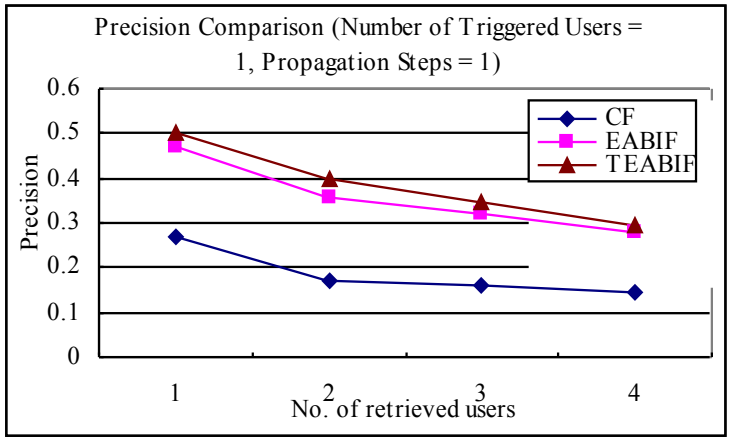
- Adoption is typically category specific
  - An early adopter of fashion may not be an early adopter of technology

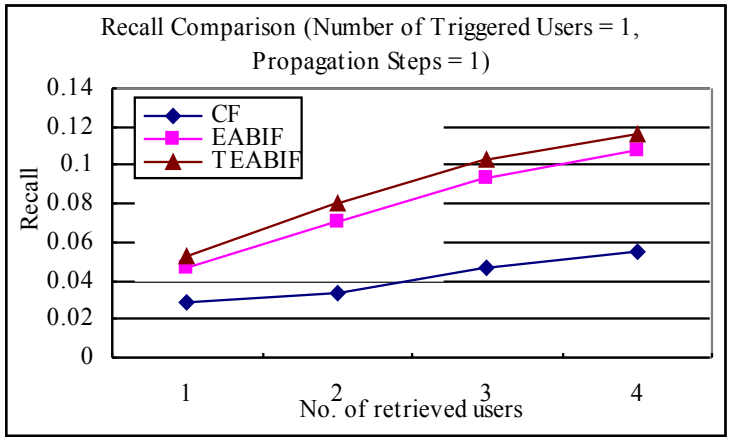


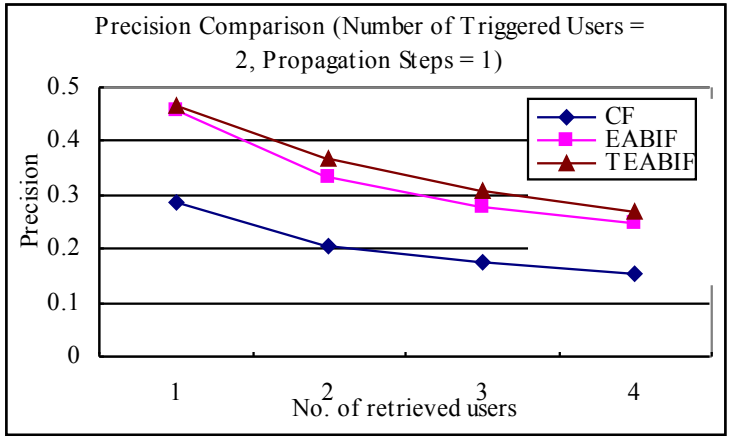
Experimental Setup

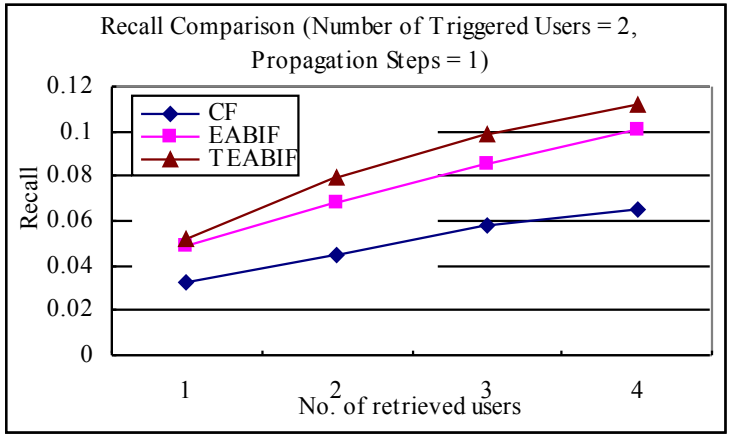
- ER dataset
  - 2004 Apr. to 2005 Apr. as training data
  - 2005 May to 2005 Jul. as test data
    - 1033 users, 586 documents
- Process
  - Construct information flow network based on the training data
  - Trigger earliest users to start the process
  - Predict who will be also interested in these documents
- Evaluation
  - Precision & Recall

Experimental Results --Recommendation Quality



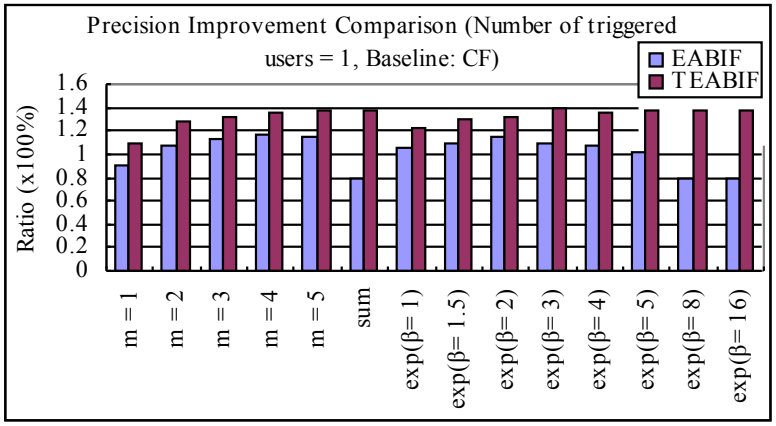


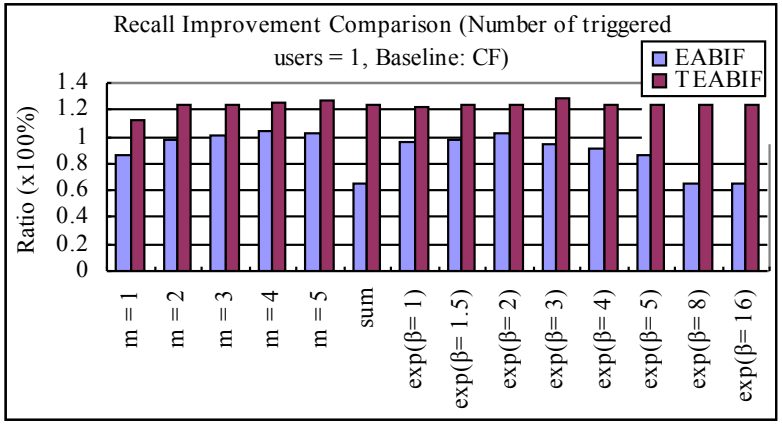


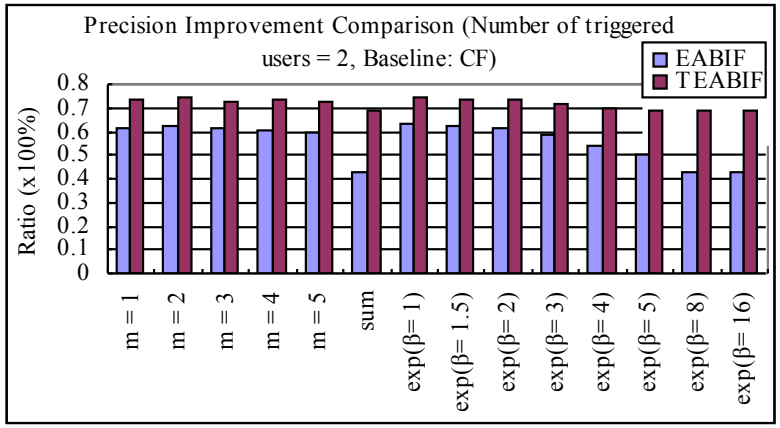


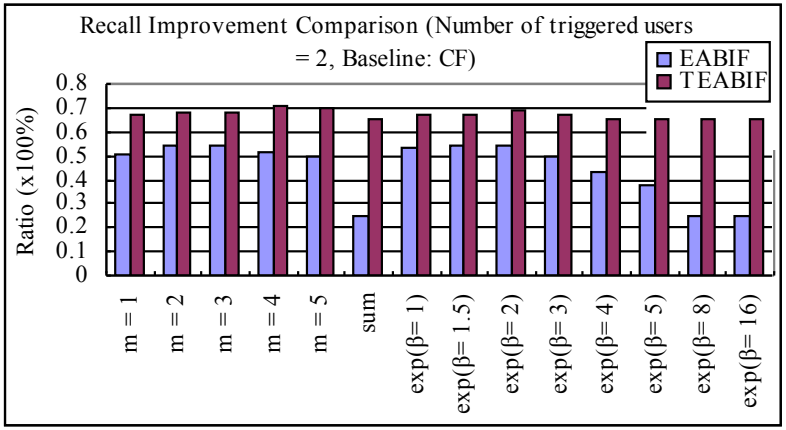
→ Comparing to Collaborative Filtering (CF),
precision: EABIF is 91.0% better, TEABIF is 108.5% better
Recall: EABIF is 87.1% better, TEABIF is 112.8% better

Experimental Results -- Propagation Performance









→ TEABIF with exponential weighted summation ($\beta = 3$) achieves the best performance:
improves 108.5% on precision and 116.9% on recall comparing to CF

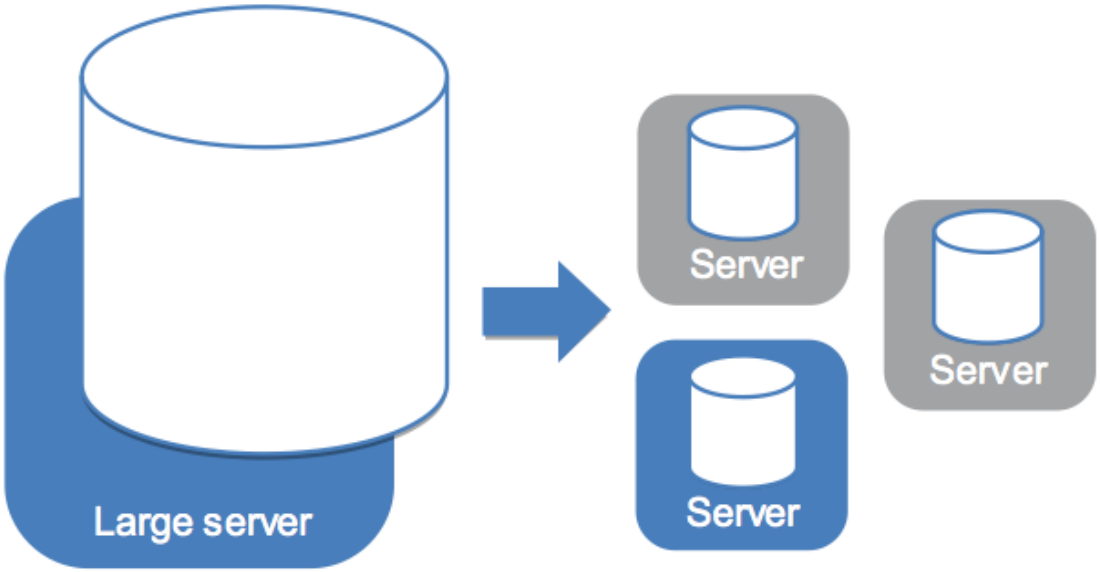
Summary

- Exploit dynamic patterns including
  - Leverage dynamic patterns from both documents and users' perspective
    - Analyzing documents accessing types
    - Predicting documents' expiration date
    - Detecting users' intentions
    - Identifying users interests evolving over time – CTC model
    - Ranking the documents adaptively – Time-sensitive Adaboost
  - Utilize users' adoption patterns
    - Information virtually flows from early adopters to late adopters
- Experimental results demonstrate
  - Dynamic factors are important for recommendations

Selected Publications

- **X. Song**, C.-Y. Lin, B. L. Tseng, and M.-T. Sun, "Modeling Evolutionary Behaviors for Community-based Dynamic Recommendation," SIAM Conf. on Data Mining, Bethesda, MD, Apr. 2006.
- **X. Song**, C.-Y. Lin, B. L. Tseng, and M.-T. Sun, "Personalized Recommendation Driven by Information Flow," ACM SIGIR, Aug. 2006.
- **X. Song**, C.-Y. Lin, B. L. Tseng and M.-T. Sun, "Modeling and Predicting Personal Information Dissemination Behavior," ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Aug. 2005.
- **X. Song**, B. L. Tseng, C.-Y. Lin, and M.-T. Sun, "ExpertiseNet: Relational and Evolutionary Expert Modeling," International Conference on User Modeling, Edinburgh, UK, Jul. 24-30, 2005.

Distributed Item-based Recommender



Distributed recommender — get co-occurrence matrix

Data:

```
1,101,5.0
1,102,3.0
1,103,2.5

2,101,2.0
2,102,2.5
2,103,5.0
2,104,2.0

3,101,2.5
3,104,4.0
3,105,4.5
3,107,5.0

4,101,5.0
4,103,3.0
4,104,4.5
4,106,4.0

5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0
```

	101	102	103	104	105	106	107
101	5	3	4	4	2	2	1
102	3	3	3	2	1	1	0
103	4	3	4	3	1	2	0
104	4	2	3	4	2	2	1
105	2	1	1	2	2	1	1
106	2	1	2	2	1	2	0
107	1	0	0	1	1	0	1

Multiply the co-occurrence matrix with user preference

	101	102	103	104	105	106	107		U3		R
101	5	3	4	4	2	2	1		2.0		40.0
102	3	3	3	2	1	1	0		0.0		18.5
103	**4**	**3**	**4**	**3**	**1**	**2**	**0**	**X**	**0.0**	**=**	**24.5**
104	4	2	3	4	2	2	1		4.0		40.0
105	2	1	1	2	2	1	1		4.5		26.0
106	2	1	2	2	1	2	0		0.0		16.5
107	1	0	0	1	1	0	1		5.0		15.5

The highest is 103 (101, 104, 105, 107 have been purchased by user 3)

Translating to MapReduce: generating user vectors

- Input files are treated as (`Long`,`String`) pairs by the framework, where the `Long` key is a position in the file and the `String` value is the line of the text file. For example, `239 / 98955: 590 22 9059`
- Each line is parsed into a user ID and several item IDs by a map function. The function emits new key-value pairs: a user ID mapped to item ID, for each item ID. For example, `98955 / 590`
- The framework collects all item IDs that were mapped to each user ID together.
- A reduce function constructors a `Vector` from all item IDs for the user, and outputs the user ID mapped to the user's preference vector. All values in this vector are 0 or 1. For example, `98955 / [590:1.0, 22:1.0, 9059:1.0]`

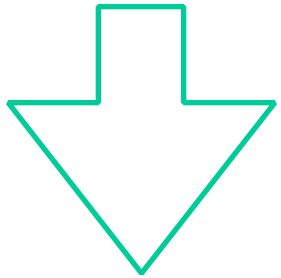
Translating to MapReduce: calculating co-occurrence

1. Input is user IDs mapped to `Vectors` of user preferences—the output of the last MapReduce. For example, `98955 / [590:1.0,22:1.0,9059:1.0]`
2. The map function determines all co-occurrences from one user's preferences, and emits one pair of item IDs for each co-occurrence—item ID mapped to item ID. Both mappings, from one item ID to the other and vice versa, are recorded. For example, `590 / 22`
3. The framework collects, for each item, all co-occurrences mapped from that item.
4. The reducer counts, for each item ID, all co-occurrences that it receives and constructs a new `Vector` that represents all co-occurrences for one item with a count of the number of times they have co-occurred. These can be used as the rows—or columns—of the co-occurrence matrix. For example,

```
590 / [22:3.0,95:1.0,...,9059:1.0,...]
```

Translating to MapReduce: matrix multiplication

```
for each row i in the co-occurrence matrix
    compute dot product of row vector i with the user vector
    assign dot product to ith element of R
```



```
assign R to be the zero vector
for each column i in the co-occurrence matrix
    multiply column vector i by the ith element of the user vector
    add this vector to R
```

Translating to MapReduce: partial products

- Input for mapper 1 is the co-occurrence matrix: item IDs as keys, mapped to columns as `Vectors`. For example, `590 / [22:3.0,95:1.0,...,9059:1.0,...]`

  The map function simply echoes its input, but with the `Vector` wrapped in a `VectorOrPrefWritable`.

- Input for mapper 2 is again the user vectors: user IDs as keys, mapped to preference `Vectors`. For example, `98955 / [590:1.0,22:1.0,9059:1.0]`

  For each nonzero value in the user vector, the map function outputs an item ID mapped to the user ID and preference value, wrapped in a `VectorOrPrefWritable`. For example, `590 / [98955:1.0]`

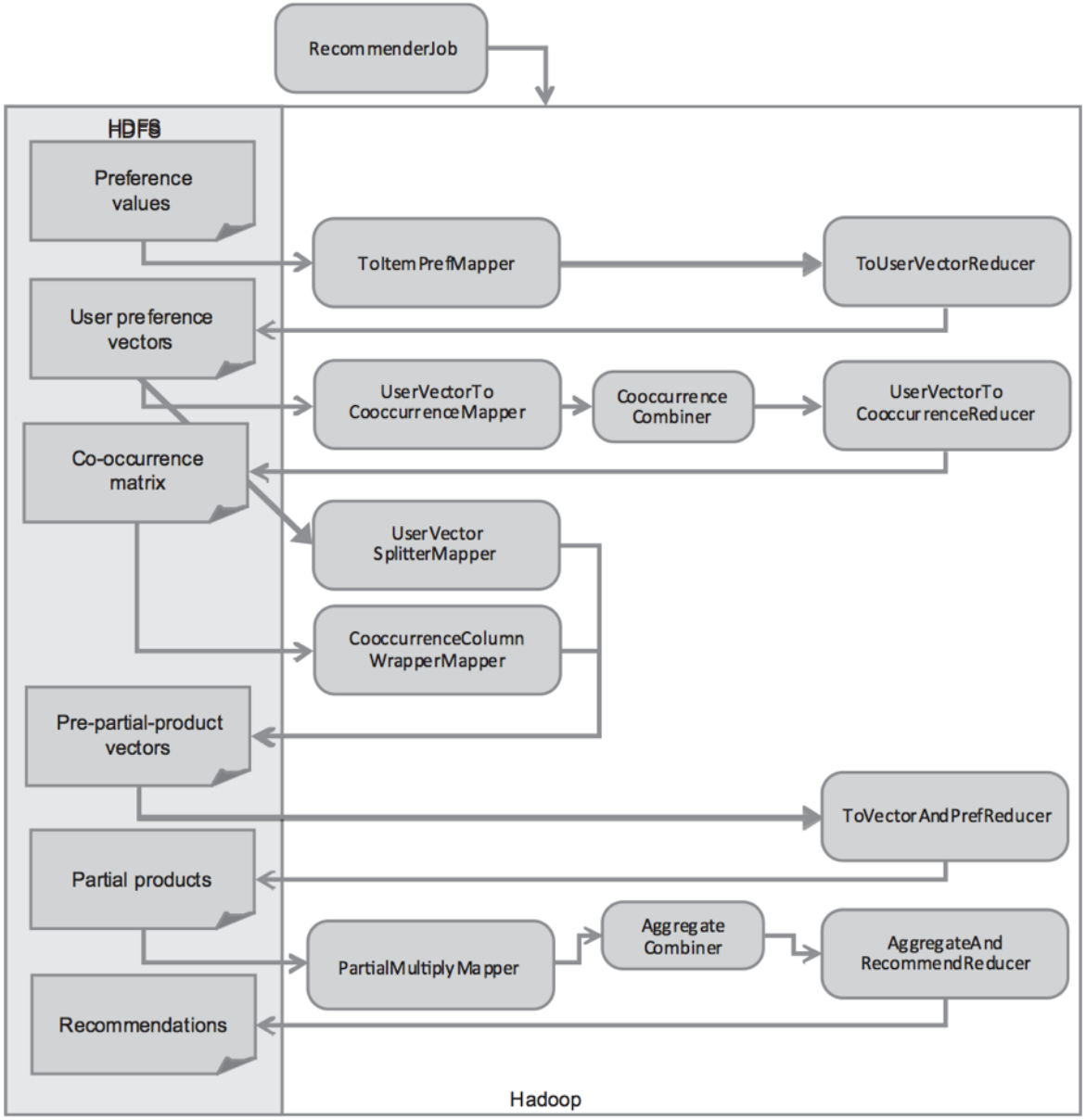
  The framework collects together, by item ID, the co-occurrence column and all user ID–preference value pairs.

  The reducer collects this information into one output record and stores it.

Translating to MapReduce: partial product II

1. Input to the mapper is all co-occurrence matrix columns and user preferences by item. For example, `590 / [22:3.0,95:1.0,...,9059:1.0,...]` and `590 / [98955:1.0]`
2. The mapper outputs the co-occurrence column for each associated user times the preference value. For example, `590 / [22:3.0,95:1.0,...,9059:1.0,...]`
3. The framework collects these partial products together, by user.
4. The reducer unpacks this input and sums all the vectors, which gives the user's final recommendation vector (call it R). For example, `590 / [22:4.0,45:3.0,95:11.0,...,9059:1.0,...]`

Running Recommender on MapReduce and HDFS



Questions?