E6893 Big Data Analytics Lecture 3:

Big Data Storage and Analytics

Ching-Yung Lin, Ph.D.
Adjunct Professor, Dept. of Electrical Engineering and Computer Science
Distinguished Research and Chief Scientist, Graph Computing, IBM Research

September 22nd, 2016
Case Study Demos — Pig, Hive and HBase by TA
Analytics 1 — Recommendation
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
PART 1 RECOMMENDATIONS ......................................................... 11
1  ▪ Meet Apache Mahout  1
2  ▪ Introducing recommenders  13
3  ▪ Representing recommender data  26
4  ▪ Making recommendations  41
5  ▪ Taking recommenders to production  70
6  ▪ Distributing recommendation computations  91

PART 2 CLUSTERING .................................................................... 115
7  ▪ Introduction to clustering  117
8  ▪ Representing data  130
9  ▪ Clustering algorithms in Mahout  145
10 ▪ Evaluating and improving clustering quality  184
11 ▪ Taking clustering to production  198
12 ▪ Real-world applications of clustering  210

PART 3 CLASSIFICATION .............................................................. 225
13 ▪ Introduction to classification  227
14 ▪ Training a classifier  255
15 ▪ Evaluating and tuning a classifier  281
16 ▪ Deploying a classifier  307
17 ▪ Case study: Shop It To Me  341

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
Motivation – Information Overload

Provide users with the information they need at the right time.
Recommender Systems: General Idea

User profile (info about the user) \(\leftrightarrow\) Set of items

comparison

Recommendation

e.g. news articles, books, music, movies, products, …

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

Preference:

Matching

Recommending

Item Stream

sports  Tech

World  World  Health  Health  …
Collaborative Filtering (CF)

Leveraging opinions of like-minded users

People with similar tastes

Given

Recommend
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

<table>
<thead>
<tr>
<th>User</th>
<th>Item</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>101</td>
<td>5.0</td>
</tr>
<tr>
<td>1</td>
<td>102</td>
<td>3.0</td>
</tr>
<tr>
<td>1</td>
<td>103</td>
<td>2.5</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>2.0</td>
</tr>
<tr>
<td>2</td>
<td>102</td>
<td>2.5</td>
</tr>
<tr>
<td>2</td>
<td>103</td>
<td>5.0</td>
</tr>
<tr>
<td>2</td>
<td>104</td>
<td>2.0</td>
</tr>
<tr>
<td>3</td>
<td>101</td>
<td>2.5</td>
</tr>
<tr>
<td>3</td>
<td>104</td>
<td>4.0</td>
</tr>
<tr>
<td>3</td>
<td>105</td>
<td>4.5</td>
</tr>
<tr>
<td>3</td>
<td>107</td>
<td>5.0</td>
</tr>
<tr>
<td>4</td>
<td>101</td>
<td>5.0</td>
</tr>
<tr>
<td>4</td>
<td>103</td>
<td>3.0</td>
</tr>
</tbody>
</table>

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!
**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"));  // Load data file

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

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

Data:

<table>
<thead>
<tr>
<th>Item 101</th>
<th>Item 102</th>
<th>Item 103</th>
<th>Correlation with user 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>5.0</td>
<td>3.0</td>
<td>2.5</td>
</tr>
<tr>
<td>User 2</td>
<td>2.0</td>
<td>2.5</td>
<td>5.0</td>
</tr>
<tr>
<td>User 3</td>
<td>2.5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>User 4</td>
<td>5.0</td>
<td>-</td>
<td>3.0</td>
</tr>
<tr>
<td>User 5</td>
<td>4.0</td>
<td>3.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

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

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

\[ \text{Similarity} = \frac{1}{1 + d} \]
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

<table>
<thead>
<tr>
<th>Variable ( X_i )</th>
<th>Position in the ascending order</th>
<th>Rank ( x_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>2</td>
<td>( \frac{2 + 3}{2} = 2.5 )</td>
</tr>
<tr>
<td>1.2</td>
<td>3</td>
<td>( \frac{2 + 3}{2} = 2.5 )</td>
</tr>
<tr>
<td>2.3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item 101</th>
<th>Item 102</th>
<th>Item 103</th>
<th>Correlation to user 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>3.0</td>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>User 2</td>
<td>1.0</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>User 3</td>
<td>1.0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>User 4</td>
<td>2.0</td>
<td>-</td>
<td>1.0</td>
</tr>
<tr>
<td>User 5</td>
<td>3.0</td>
<td>2.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Item 101</th>
<th>Item 102</th>
<th>Item 103</th>
<th>Item 104</th>
<th>Item 105</th>
<th>Item 106</th>
<th>Item 107</th>
<th>Similarity to user 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>User 2</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>0.75</td>
</tr>
<tr>
<td>User 3</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0.17</td>
</tr>
<tr>
<td>User 4</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>0.4</td>
</tr>
<tr>
<td>User 5</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0.5</td>
</tr>
</tbody>
</table>

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.

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

<table>
<thead>
<tr>
<th></th>
<th>Item 101</th>
<th>Item 102</th>
<th>Item 103</th>
<th>Item 104</th>
<th>Item 105</th>
<th>Item 106</th>
<th>Item 107</th>
<th>Similarity to user 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.90</td>
</tr>
<tr>
<td>User 2</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>0.84</td>
</tr>
<tr>
<td>User 3</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>0.55</td>
</tr>
<tr>
<td>User 4</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>0.16</td>
</tr>
<tr>
<td>User 5</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0.55</td>
</tr>
</tbody>
</table>
Performance measurements

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

<table>
<thead>
<tr>
<th></th>
<th>Item 1</th>
<th>Item 2</th>
<th>Item 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual</td>
<td>3.0</td>
<td>5.0</td>
<td>4.0</td>
</tr>
<tr>
<td>Estimate</td>
<td>3.5</td>
<td>2.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Difference</td>
<td>0.5</td>
<td>3.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Average difference</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$(0.5 + 3.0 + 1.0) / 3 = 1.5$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Root-mean-square</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\sqrt{(0.5^2 + 3.0^2 + 1.0^2) / 3} = 1.8484$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance measurements

```java
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

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

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

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

    performance

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PearsonCorrelationSimilarity</td>
<td>0.75</td>
</tr>
<tr>
<td>PearsonCorrelationSimilarity + weighting</td>
<td>0.75</td>
</tr>
<tr>
<td>EuclideanDistanceSimilarity</td>
<td>0.76</td>
</tr>
<tr>
<td>EuclideanDistanceSimilarity + weighting</td>
<td>0.78</td>
</tr>
<tr>
<td>TanimotoCoefficientSimilarity</td>
<td>0.77</td>
</tr>
<tr>
<td>LogLikelihoodSimilarity</td>
<td>0.77</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Item 101</th>
<th>Item 102</th>
<th>Item 103</th>
<th>Item 104</th>
<th>Item 105</th>
<th>Item 106</th>
<th>Item 107</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item 101</td>
<td>-0.833</td>
<td>0.875</td>
<td>0.25</td>
<td>0.75</td>
<td>-0.5</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>Item 102</td>
<td>0.333</td>
<td>0.25</td>
<td>0.5</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item 103</td>
<td></td>
<td>0.167</td>
<td>1.5</td>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item 104</td>
<td></td>
<td></td>
<td>0.0</td>
<td>-0.25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item 105</td>
<td></td>
<td></td>
<td></td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item 106</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item 107</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Difference values from the example

Slope-One got a result of near 0.65 on the GroupLens data
Other recommenders — SVD recommender

```java
new SVDR recommender(model, new ALSWRFFactorizer(model, 10, 0.05, 10))
```

- number of features
- number of training step
- lambda: factor for regularization

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.

```java
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
Innovator

People with similar tastes

Early adopter

Early majority

Late majority

Laggard

Influence is not symmetric!
Scheme Overview

- Leverage the uneven influence

EABIF: Early Adoption based Information Flow Network

Application: Personalized Recommendation
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

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 $\sum_j F_{ij} = 1$
    - Make the Markov chain irreducible $\overline{F}_{ij} \neq 0, \ 1 \leq i, j \leq N$
Information Propagation Models

1. Summation of various propagation steps

\[ F_{if(m)} = \left( \bar{F} + \bar{F}^{(2)} + \bar{F}^{(3)} + \ldots + \bar{F}^{(m)} \right) / m \]

2. Direct summation

\[ F_{if(d)} = \frac{\bar{F} \cdot \left( I - \bar{F}^{(N-1)} \right)}{I - \bar{F}} \]

3. Exponential weighted summation

\[ F_{if(exp)} = \left( \beta \bar{F} \right)^0 + \frac{1}{2!} \left( \beta \bar{F} \right)^2 + \frac{1}{(N-1)!} \left( \beta \bar{F} \right)^{(N-1)} + \exp(-\beta) \]

\( N \): number of the nodes
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
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

Precision Improvement Comparison (Number of triggered users = 1, Baseline: CF)

Recall Improvement Comparison (Number of triggered users = 1, Baseline: CF)

- 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


Distributed Item-based Recommender

Large server

Server

Server

Server
Distributed recommender — get co-occurrence matrix

<table>
<thead>
<tr>
<th></th>
<th>101</th>
<th>102</th>
<th>103</th>
<th>104</th>
<th>105</th>
<th>106</th>
<th>107</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>102</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>103</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>104</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>105</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>106</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>107</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Multiply the co-occurrence matrix with user preference

<table>
<thead>
<tr>
<th></th>
<th>101</th>
<th>102</th>
<th>103</th>
<th>104</th>
<th>105</th>
<th>106</th>
<th>107</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>102</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>103</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>104</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>105</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>106</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>107</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>U3</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>4.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>4.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>5.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ X \times \begin{pmatrix} 2.0 \\ 0.0 \\ 0.0 \\ 4.0 \\ 4.5 \\ 0.0 \\ 5.0 \end{pmatrix} = \begin{pmatrix} 40.0 \\ 18.5 \\ 24.5 \\ 40.0 \\ 26.0 \\ 16.5 \\ 15.5 \end{pmatrix} \]

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 constructs 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 $i$th 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 $i$th 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 / \([22:3.0, 95:1.0, \ldots, 9059:1.0, \ldots]\) and 590 / \([98955:1.0]\).

2. The mapper outputs the co-occurrence column for each associated user times the preference value. For example, 590 / \([22:3.0, 95:1.0, \ldots, 9059:1.0, \ldots]\).

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 / \([22:4.0, 45:3.0, 95:11.0, \ldots, 9059:1.0, \ldots]\).
Running Recommender on MapReduce and HDFS

- **HDFS**
  - Preference values
  - User preference vectors
  - Co-occurrence matrix
  - Pre-partial-product vectors
  - Partial products
  - Recommendations

- **RecommendationJob**
  - **ToItemPrefMapper** → **ToUserVectorReducer**
  - **UserVectorToCooccurrenceMapper** → **CooccurrenceCombiner** → **UserVectorToCooccurrenceReducer**
  - **UserVectorSplitterMapper**
  - **CooccurrenceColumnWrapperMapper**
  - **ToVectorAndPrefReducer**
  - **PartialMultiplyMapper** → **AggregateCombiner** → **AggregateAndRecommendReducer**
Questions?
E6893 Big Data Analytics:

*Demo Session for HW I*
Part I: Pig installation and Demo

Pig is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs.
1. Installation of Pig:
https://pig.apache.org/docs/r0.7.0/setup.html

Download pig and run following sentence:

```bash
export PATH=/<my-path-to-pig>/pig-n.n.n/bin:$PATH
```
2014-10-02 00:44:13.504 java[56934:1903] Unable to load realm info from SCDynamicStore
2014-10-02 00:44:13,767 [main] WARN org.apache.hadoop.util.NativeCodeLoader - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
2014-10-02 00:44:14,295 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS

2. Running pig in local mode:

```
pig -x local
movies = LOAD '/Users/Rich/Documents/Courses/Fall2014/BigData/Pig/movies_data_data.csv' USING PigStorage(',') as (id,name,year,rating,duration);
DUMP movies
```
Filter:

List the movies that were released between 1950 and 1960

movies_1950_1960 = FILTER movies BY (float)year>1949 and (float)year<1961;

Foreach Generate:

List movie names and their duration (in minutes)

movies_name_duration = foreach movies generate name, (float)duration/3600;

store movies_name_duration into '/Users/Rich/Desktop/Demo/movies_name_duration';
Order:

List all movies in descending order of year

movies_year_sort =order movies by year desc;

store movies_year_sort into '/Users/Rich/Desktop/Demo/movies_year_sort';
3. Running pig with HDFS:

```
pig
```

Run HDFS first:

```
ssh localhost

cd /usr/local/Cellar/hadoop/2.5.0

sbin/start-dfs.sh
```
3. Upload file to HDFS:

   Make a directory:

   `bin/hdfs dfs -mkdir /PigSource`

   Upload a file:

4. Run pig in grunt with HDFS:

```pig
movies = LOAD '/PigSource/movies_data.csv' USING PigStorage(',') as (id,name,year,rating,duration);

DUMP movies
```
5. Run .pig file with HDFS:

```
pig

pig /Users/Rich/Documents/Courses/Fall2014/BigData/Pig/run1.pig
```
Part II: Hbase installation and Demo

HBase is an open source, non-relational, distributed database modeled after Google's Big Table and written in Java.

It runs on top of HDFS, and can serve as input and output for MapReduce jobs run in Hadoop.

Access through Java API and Pig.
Hbase Configuration:

Download and configure Hbase by following the instruction online.

To run Hbase, under your Hbase path:
$bin/start-hbase.sh
Enter shell
$bin/hbase shell

Also visit localhost:60010 to check Hbase webUI
Hbase Command:

Create:
hbase>create 'table', 'cf'
hbase>put 'table', 'r1', 'cf:c1', 'value1'
hbase>scan 'table'

Also we can do count, get, delete and drop. etc much like other DB systems.
Part III: Hive installation and Demo
Simple demo for Hive

Install steps:

**Mac:** you can first install brew

```
$ brew install Hive
```

**Linux:** cd ~/Downloads

```
wget http://mirror.cc.columbia.edu/pub/software/apache/hive/hive-0.13.1/apache-hive-0.13.1-bin.tar.gz
cp apache-hive-0.13.1-bin.tar.gz ~
cd ~
tar zxvf apache-hive-0.13.1-bin.tar.gz
mv apache-hive-0.13.1-bin hive
cd hive
cd bin
./hive
```
Simple demo for Hive

Step 1. start Hadoop, create a file under /user/
yourname

    hadoop fs -mkdir test

Step 2. put your dataset into hdfs

    hadoop fs -put (your dataset path) test
    hadoop fs -ls test

Found 1 items
rw-r--r--  1 huanglin supergroup  204 2014-09-30 22:49 test/client.txt
Simple demo for Hive

Step 3. create the table (SQL) in Hive shell
hive

```
hive> create table client(
  > name String,
  > gender String,
  > age INT,
  > job String,
  > nation String,
  > tele INT)
  > row format delimited
  > fields terminated by 't'
  > lines terminated by '\n'
  > stored as textfile;
```

```
hive> select * from client;
OK
John  male  17  students  USA    111111
Henry male  25  sportsman UK     222222
Kim   female  29  actor     Korea 33333
Zhu   female  21  graduate  China 444444
Alex  male  47  professor  France 555555
Luca  male  30  banker     Swiss 666666
Time taken: 0.038 seconds, Fetched: 6 row(s)
```

```
OK
Time taken: 0.049 seconds
```

Step 4. store the data into table
load data inpath '/user/yourname/test' into table client;
You can see your data has already been put into the table
Simple demo for Hive

1. select * from client where name='Henry';

Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
Kill Command = /usr/local/Cellar/hadoop/2.5.1/libexec/bin/hadoop job -kill job_1412113170187_0003
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 0
2014-09-30 22:53:03,624 Stage-1 map = 0%, reduce = 0%
2014-09-30 22:53:09,947 Stage-1 map = 100%, reduce = 0%
Ended Job = job_1412113170187_0003
MapReduce Jobs Launched:
Job 0: Map: 1 HDFS Read: 417 HDFS Write: 34 SUCCESS
Total MapReduce CPU Time Spent: 0 msec
OK
Henry male 25 sportsman UK 222222
Time taken: 15.673 seconds, Fetched: 1 row(s)

2. select avg(age) from client where gender='male';

OK
29.75
Time taken: 21.521 seconds, Fetched: 1 row(s)
Use JDBC(JAVA) to access data in Hive Server
Other choice: Python, PHP etc.

```java
Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/default", "", "");
Statement stmt = con.createStatement();
String tableName = "client1";
stmt.execute("drop table if exists " + tableName);
stmt.execute("create table " + tableName + " (name String, gender String, age INT, job String,
    + "nation String, tele INT) row format delimited fields terminated by '\t' lines"
    + "terminated by '\n' stored as textfile");
System.out.println("Create table success!");
String filepath = "/user/huanglin/hive";
String sql = "load data inpath " + filepath + " into table " + tableName;
System.out.println("Running: " + sql);
ResultSet res = stmt.executeQuery(sql);
sql = "select name,nation from " + tableName + " where age>25";
System.out.println("Running: " + sql);
res = stmt.executeQuery(sql);
while (res.next()) {
    System.out.println(res.getString(1) + "\t"
        + res.getString(2));
}
sql = "select * from " + tableName + " where nation='USA'";
System.out.println("Running: " + sql);
res = stmt.executeQuery(sql);
while (res.next()) {
    System.out.println(res.getString(1) + " " + res.getString(2) + " " + String.valueOf(res.getInt(3)) +
        " " + res.getString(4)+" " +res.getString(5)+" " +String.valueOf(res.getInt(6)));
```
Open the Hive (Starting Hive Thrift Server)
serverhive --service hiveserver -p 10000

URL: jdbc:hive://localhost:10000/default

Result in the console
Simple demo for Hive

Example. Jetty Server with Jersey, restful web service

Create simple API (Http request, XML/JSON format output)

```java
@Path("/Api")
public class service {

@Path("/getinfo/{name}")
@GET
@Produces(MediaType.APPLICATION_XML)
public client getinfo(@PathParam("name") String name) throws SQLException {
    try {
        Class.forName("org.apache.hadoop.hive.jdbc.HiveDriver");
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
        System.exit(1);
    }
    Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/default", ",", "");
    Statement stmt = con.createStatement();
    String tableName = "client1";
    stmt.execute("drop table if exists " + tableName);
    stmt.execute("create table " + tableName + " (name String, gender String, age INT, job String, "
        + "nation String, tele INT) row format delimited fields terminated by '\\t' lines 
        + "terminated by '\n' stored as textfile");
    String filepath = "/user/huanglin/hive";
    String sql = "load data inpath " + filepath + " into table " + tableName;
    ResultSet res = stmt.executeQuery(sql);
    sql = "select * from client1 where name='"+name+'"";
    System.out.println("Running: " + sql);
    res = stmt.executeQuery(sql);
    while (res.next()) {
        client user = new client();
        user.setName(res.getString(1));
        user.setGender(res.getString(2));
        user.setAge(res.getInt(3));
        user.setJob(res.getString(4));
        user.setNation(res.getString(5));
        user.setTele(res.getInt(6));
        return user;
    }
    return null;
}
```

Simple demo for Hive

Example. Jetty Server with Jersey, restful web service

```java
public class server {
    public static void main(String[] args) throws Exception {
        Server server = new Server(7777);
        ServletContextHandler context = new ServletContextHandler(ServletContextHandler.SESSIONS);
        context.setContextPath("/");
        context.setHandler(context);
        ServletHolder jerseyServlet = context.addServlet(org.glassfish.jersey.servlet.ServletContainer.class, "/");
        jerseyServlet.setInitOrder(0);
        jerseyServlet.setInitParameter("jersey.config.server.provider.classnames", "rest.service");
        server.start();
        server.join();
    }
}
```

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```xml
<client>
    <age>47</age>
    <gender>male</gender>
    <job>professor</job>
    <name>Alex</name>
    <tele>555555</tele>
</client>
```