Module 13: Machine Learning – Collaborative Filtering

Machine Learning (ML) is a branch of Artificial Intelligence (AI). Machine Learning algorithms encode techniques to automatically learn from data. Essentially, they examine data to predict how new data is to be viewed. While there are many ways to categorize ML algorithms, we will categorize them into 3 main categories:

1. Recommendation – These algorithms ask questions like, "Which new products should I recommend to you based on past purchases?"
2. Classification – These algorithms determine the category to which some piece of data belongs. They answers questions like, "Is this a picture of a car or a mailbox?"
3. Clustering – These algorithms determine the best way to organize data. They answer questions like, "Is this piece of data more like the first set of data or the second?"

Machine Learning addresses both the representation of the data as well as the algorithms that operate on the data. Ultimately, these algorithms generalize the current data to predict accurately future occurrences of data.

Machine Learning and Data Mining are not quite the same, but they overlap significantly. Machine Learning is prediction focused where predictions are based on properties examined in the data used to train the algorithms. Data Mining is discovery focused, where algorithms find new and unknown properties of the data. Data mining uses many machine learning methods, but with the focus of discovering new information in data. All these concepts will be developed over the next few modules.

Apache Mahout

Mahout (rhymes with *about*) is a scalable library of machine learning tools that operate on top of Hadoop. Hadoop was named after Doug Cutting's son's toy elephant. Mahout is the Hindi word for *elephant driver*. Mahout is available on http://mahout.apache.org/. The general categories of algorithms are Collaborative Filtering (Recommenders), Clustering, Classification, and Data Mining.

Mahout started in 2008 as a subproject of Apache Lucene (https://lucene.apache.org/). Lucene provides search, text mining, and information retrieval algorithms. Many of the text algorithms require clustering and classification. Soon, however, the clustering and classification algorithms took on their own lives. These algorithms in addition to some collaborative filtering algorithms were merged into a single project and the group of algorithms was named Mahout. The algorithms are well tested for both small and large-scale machine learning. In addition, Mahout provides an incubator for new techniques.

Collaborative Filtering (Recommenders) have two flavors. The first is people based. The algorithms of this class examine your actions, compare your actions to other people like you, determine what those people liked that you have not tried yet, and

recommend those items to you. The second flavor is item based. The algorithms of this class bypass other people. They focus on the items that are similar to the items you like and recommend other items like those you like. Example of services that employ collaborative filtering are Amazon.com (recommends items that you are likely to want to purchase), Netflix.com (recommends new movies), and LinkedIn (recommends new contacts and groups).

Clustering algorithms group many items into clusters that share similar properties. For example, Google News automatically groups articles by topic to present a logical cluster of stories.

Classification algorithms determine whether items belong to known categories. For example classifiers may automatically check for spam email, identify weapons from images, classify accents of callers in voicemail messages, and detect insider threats based on network activities.

This module will focus on Collaborative Filtering. The next module will focus on Clustering and Classification.

What is needed to Set Up Mahout

This module does not require programming in Mahout. It will only show examples of Mahout code. However, to set up a Mahout environment, the following needs to be downloaded and installed.

- **Install Java and an IDE**
  - Need Java 6: http://www.oracle.com/technetwork/java/
  - Eclipse: http://www.eclipse.org/
  - or Netbeans: http://netbeans.org/
- **Maven**
  - Manages code dependencies, compiles code, packages releases, generates documentation
  - http://maven.apache.org/
  - Eclipse
    - *Maven is part of Eclipse already*
    - *Need to install m2eclipse plugin http://www.eclipse.org/m2e/*
- **Mahout**
  - https://cwiki.apache.org/confluence/display/MAHOUT/Downloads
- **Hadoop**
  - http://hadoop.apache.org/common/releases.html
  - http://hapoop.apache.org/common/docs/current/single_node_setup.html

This information is provided for the reader who wants to try out the concepts presented in this and the next modules.

Collaborative Filtering Premises

There are two premises behind collaborative filtering that make it successful. The first premise is that people will probably like new items that are similar to the items

they already have. This is known as *item-based* collaborative filtering. Some examples are:

- If Sam likes reading sci-fi books, he will like other sci-fi books
- If George likes action movies, he will likely want to see the newest action movie
- If Sally likes country-western music, she will probably want to hear the newest country-western hit
- If Mary likes vacationing at beach resorts, she may be interested in a new beach resort

The second premise is that people will probably like new items that people similar to them like. This is known as *user-based* collaborative filtering. Some examples are:

- Teenage girls living in NYC will likely want the latest NYC teenage girl styles
- Math professors will likely read the same journals as other math professors

There is another type of collaborative filtering style called *content-based* collaborative filtering. This approach looks at the properties of an item and recommends items with similar properties. Some examples are:

- Fred likes books that discuss the Tour de France (content). He will probably like other books that discuss the Tour de France.
- Vicky likes movies about surfing (content). She will probably like other movies about surfing.

However, it is difficult to set up a generic framework for this type of collaborative filtering. There are so many aspects about books, movies, cars, etc. That it is difficult to determine which are important. Furthermore, content recommenders for one class of item (eg. vacation spots) are very different from recommenders for different classes of items (eg., paintings).

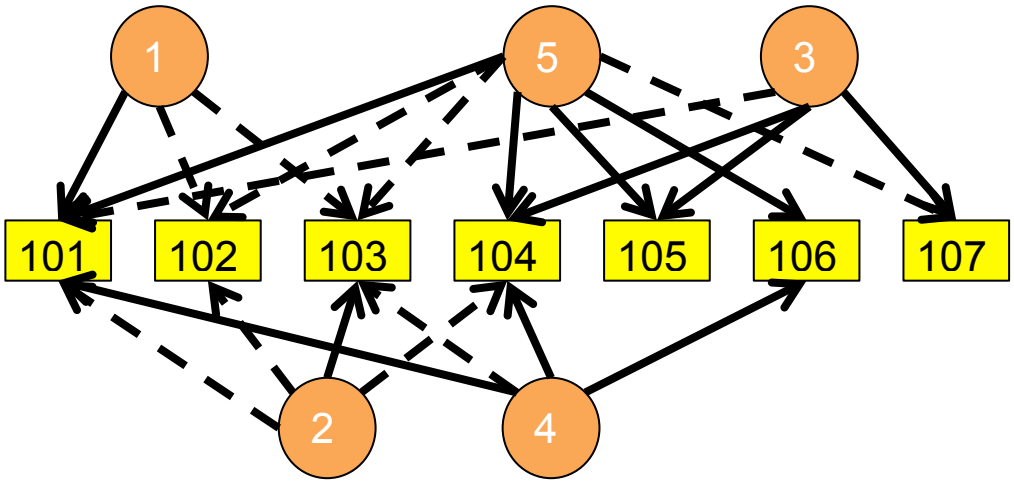
Mahout Recommender Engine

The data structure for Mahout Recommender Engines is very straightforward. It consists of three columns: UserID, ItemID, Preference (on some scale). For example:

User ID	Item ID	Preference
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	106	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

The question is which users are similar to each other? One way to visualize this is via the following image. The circles represent users and the rectangles represent the items. The solid lines represent strong preferences and the dotted lines represent weak preferences.



By examining the relationships, one can determine that

- 1 and 5 are similar (similar preferences for 101, 102, 103)
- 1 and 4 seem similar (similar preferences for 101, 103)
- 1 and 2 are almost opposite (opposite preferences for 101, 103)
- 1 and 3 are almost independent of each other
- 4 and 5 are almost identical (similar preferences for 101, 103, 104, 106)

One can readily see that finding similarities and dissimilarities are cognitively difficult for a small set of data. It is almost impossible to determine with a large set of data points.

Creating a Recommender in Mahout

Mahout reduces the problem of creating a Recommender Engine to four steps.

- Read in the data model
- Choose the algorithm for determining similarity
- Choose the neighborhood algorithm
- Build the recommender

Then, the Mahout recommender is ready to run and to make recommendations!

The first function call reads the data model. The DataModel object is an interface for multiple data models. It gets the user, item and preference information. In FileDataModel, we assume that the input file has three columns: UserID, ItemID, Preference (optional), and Timestamp (optional). Delimiters in the file are commas or tabs.

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

The second function call identifies a UserSimilarity algorithm. These algorithms must return values between -1.0 and 1.0. In this case, we choose a popular similarity algorithm called the Pearson Correlation algorithm. We will discuss the Pearson Correlation in more detail later, but it is basically sensitive to the linear relationship between two variables. The parameter is the data model.

```
UserSimilarity similarity =
   new PearsonCorrelationSimilarity (model);
```

The third function is the neighborhood function that identifies a neighborhood of closest points. We will discuss this later as well. For this example, we select the NearestNUserNeighborhood, which finds the nearest N neighbors based on the data model and similarity.

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

The third function is the recommender function. This can be one of GenericBasedRecommender, ItemBasedRecommender and UserBasedRecommender. These all return a list for iteration based on the model, similarity, and neighborhood.

```
Recommender recommender = new GenericUserBasedRecommend (model,
                                  neighborhood, similarity);
recommender.recommend(1 /* user 1 */, 1 /* one item */);
```

Putting the code together, we get:

```
Class RecommenderIntro {
    public static void main(String[] args) throws Exception {
     /* Set the data model */
       DataModel model = new FileDataModel(new File("intro.csv"));

    /* Set the similarity function */
    UserSimilarity similarity =
         new PearsonCorrelationSimilarity(model);
    /* Set the neighborhood threshold. In this case, any measure */
    /* above 2 passes. */
```

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

        /* Put it all together. */
        Recommender recommender =
            new GenericUserBasedRecommender(model, neighborhood,
                                            similarity);

        /* Call the recommend function for item with ID = 3 */
        /* and get the 10 nearest neighbors. */
        List<RecommendedItem> recommendations =
            recommender.recommend(3, 10);
        for (RecommendedItem recommendation : recommendations) {
            System.out.println(recommendation);
        }
        }
    }
```

The results are:

RecommendedItem [item:104, value:4.257081]

Why 104? Well, item 107 was liked by user 3, who is very different from user 1. User 1 is similar to users 4 and 5. The preference for 104 is a bit higher than for 106. Both users 4 and 5 liked 106 at 4.0. However, user 4 liked 104 at 4.5 and user 5 liked 105 at 4.0.

Why 4.257081? We have to examine the algorithm more closely. However, 4.3 is a strong estimate of user 1's liking of item 401.

Identifying the Best Recommender

Mahout provides an evaluator capability to test different recommenders. It is hard to know a priori which is the best. Different recommenders are better for different data sets. You basically have to try them out. We will cover a few of these soon.

To test recommenders, one should break the dataset into real and test sets. Where the number of real data items far exceed the number of test items. Then determine which recommender best predicts the data. The evaluator code for Recommenders looks like this:

```
RandomUtils.useTestSeed(); // generates repeatable results
DataModel model = new FileDataModel(new File("intro.csv"));
// One of 3 evaluators implemented in Mahout
RecommenderEvaluator evaluator =
   new AverageAbsoluteDifferenceRecommenderEvaluator();
RecommenderBuilder builder = new RecommenderBuilder() {
   @Override public Recommender buildRecommender(
                                              DataModel model)
              throws TasteException {
         UserSimilarity similarity =
              new PearsonCorrelationSimilarity(model);
         UserNeighborhood =
```

```
                new NearestNUserNeighborhood(2, similarity, model);
            return new GenericUserBasedRecommender(
                            model, neighborhood, similarity);
        }
    };

    // Discussed shortly
    Double score = evaluator.evaluate(builder, null, model, 0.7, 1.0);
    System.out.println(score);
```

The evaluator call is defined as:

```
double evaluate(RecommederBuilder builder,
           DataModelBuilder       modelBuilder,
           DataModel        model,
           double           trainingPercentage,
           double           evaluationPercentage);
```

This call evaluates the quality of a Recommender with respect to the DataModel. Lower values are better and a score of 0 is a perfect match! Typically the *trainingPercentage* is 90% of the data. The rest of the data is used to see how well the predicted values match the real values for a given recommender. The *evaluationPercentage* is the percent of data model users that are involved in the evaluation. For large datasets, it often makes sense to perform the evaluation based on a small percentage of the data. Furthermore, *trainingPercentage* and *evaluationPercentage* are not related. They do not have to add up to 1.0.

Representing Preference Data

The data abstraction for preference data is <userID, itemID, value>. Typically, this is 8 bytes for the userID, 8 bytes for the itemID, 4 bytes for the value, and 28 bytes for the object overhead. This really needs 20 bytes, but it stores 48 bytes. Mahout employs a more efficient data model that can save 400% space! It uses a linked list to represent item IDs and preferences.

There are a few classes that are used with this data model.

1. **FastMap <Key,Value>** is usually faster than Java's HashMap and it saves space as well
2. **FastByIDMap <Value>** is used for the preference array
3. **FastIDSet** is used for sets of data and it does not use Map for implementation

Here is example code:

```
DataModel model = new GenericBooleanPrefDataModel(
```

```
   // Xlates FastIdSet to PreferenceArray
   GenericBooleanPrefDataModel.toDataMap(
         new FileDataModel(new File("ua.base"))));
RecommenderEvaluator evaluator =
   new AverageAbsoluteDifferenceRecommenderEvaluator();
RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
   ... // same as before
}
DataModelBuilder modelBuilder = new DataModelBuilder() {
   public DataModel buildDataModel (
               FastByIDMap<PreferenceArray> trainingData) {
         return new GenereicBooleanPrefDataModel(
               GenericBooleanPrefDataModel.toDataMap(trainingData));
   }
};
Double score =
   evaluator.evaluate(recommenderBuilder,modelBuilder,model,.9,1);
System.out.println(score);
```

Writing a Recommender

User-based recommenders must find items that similar users have purchased. For every item that a specific user has not purchased yet, the algorithm explores other users that have this item, finds the similarity between this user and the other users based on items they have purchased, and then identifies the other users' preference for the item. The weighted preferences for the item are averaged weighted by the users' similarities. If this user, u, is similar to another user, v, then v's preference for the item is weighted heavily. If u is dissimilar to v, than v's preference for the item is weighted lightly. This is an algorithm that implements this concept.

```
for each item i that u has no preferences for yet {
   for each 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
               in a running average;
   }
}
return the top items ranked by weighted average;
```

This algorithm is inefficient because it must search the entire space of users. For a large online company, this can be very time consuming, especially if this is repeated for all users! A more efficient approach is the find a neighborhood of similar users first. Then compare the user to similar users in the neighborhood.

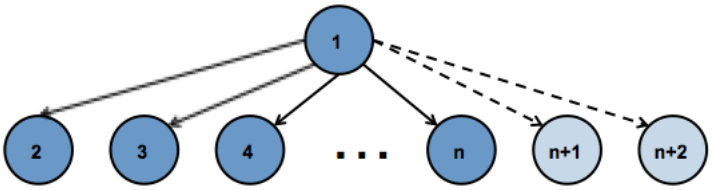
```
for each other user w {
   compute a similarity between u and w;
   keep the top users ranked by similarity in neighborhood n;
}

for each item i that some user in n has but u has no preferences yet
{
   for every other user v in n that has a preference for i {
         compute a similarity s between u and v;
```

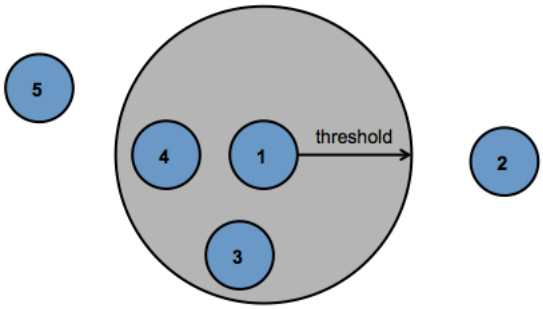
```
            incorporate v's preference for i weighted by s
                  in a running avg.;
      }
   }
   return the top items, ranked by weighted average;
```

Much fewer comparisons need be done!

What is the best-sized neighborhood? For each data set and each problem, experimentation must be done. Too small of a neighborhood does not pick up enough similar users to make good recommendations. Too large of a neighborhood may pick up too many dissimilar neighbors.



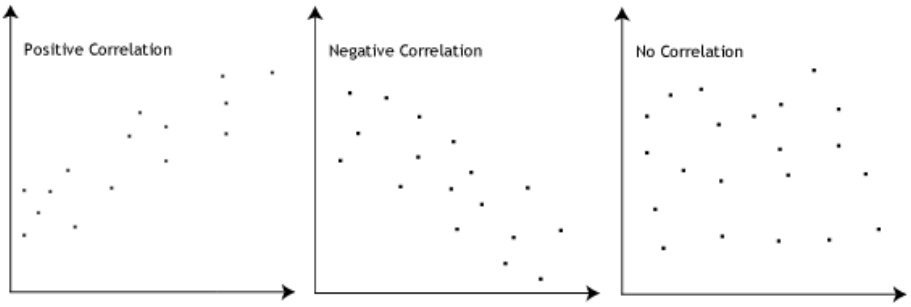
To determine a threshold, pick a similarity index between -1 and 1. The higher the similarity, the better.



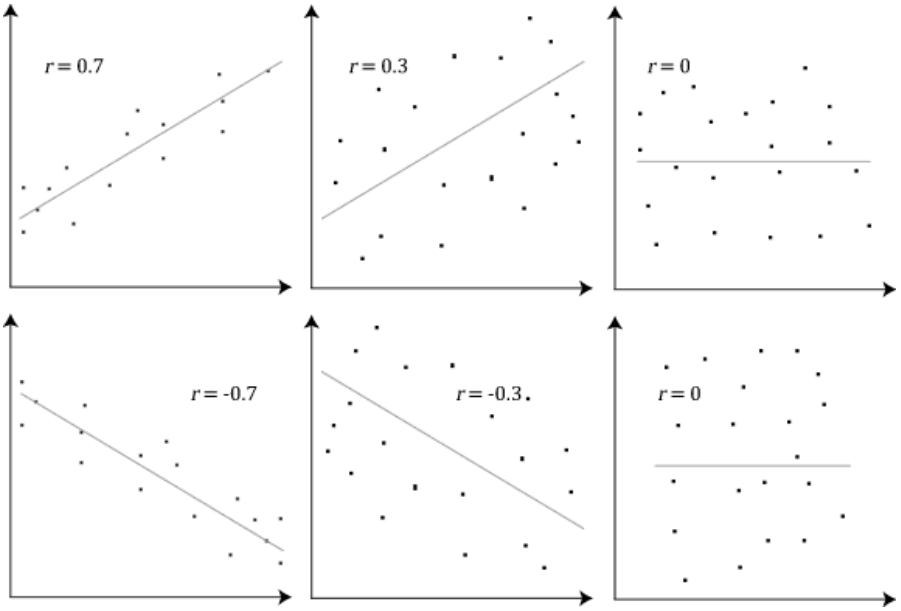
Similarity Metrics

The **Pearson Correlation** determines trending between two individuals. It seeks to determine of their preferences vary in the same direction.

Imagine a bunch of points on a graph. Their positive correlation indicates a positive Pearson correlation and a negative correlation indicates a negative Pearson correlation.



The greater the points line up, the higher the correlation.



The formal equation for the Pearson Correlation is:

$$\sum_{(p \cdot prefs)}(x_p * y_p) \,/\, sqrt\,[\, \sum_{(p \cdot prefs)}(x_p^2) * \sum_{(p \cdot prefs)}(y_p^2)\,]$$

This is equivalent to the common Cosine similarity metric that determines how close two vectors are.

$$\cos(\Theta) = A \cdot B \,/\, ||A||\,||B||$$

$$A \cdot B = \sum_{(p \cdot prefs)}(x_p * y_p)$$
$$||A||\,||B|| = sqrt\,[\, \sum_{(p \cdot prefs)}(x_p^2) * \sum_{(p \cdot prefs)}(y_p^2)\,]$$

There are some weaknesses of the Pearson Correlation. It does not consider the number of common preferences. Two users who bought two common items is treated the same as two users who bought 100 common items. This is because the cosine measure is unitless. It also shows no correlation if only one item overlaps two users. The users may not be very correlated anyway, but it would still be good to measure some correlation among these users. Finally, it only considers that users have the same preference weight for all items. This is unlikely in practice.

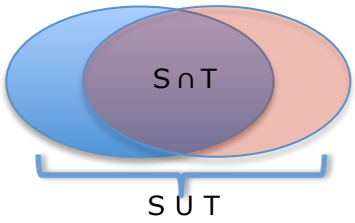
Another correlation metric is the **Euclidean Distance**. Users with similar preferences are geometrically closer to each other. The equation is just like the geometric equation:

$$\text{distance} = \text{sqrt}\left[\sum_{(p \,.\, \text{prefs})} (x_p - y_p)^2 \right]$$

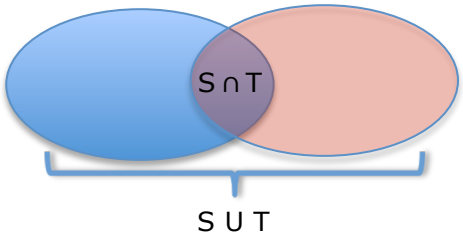
Normalizing so that close preferences approach 1 and distant preference approach 0:

$$\text{metric} = 1 / (1 + \text{distance})$$

The **Tanimoto Coefficient** or the **Jaccard Index** treats preferences as sets. If the intersection of the set of items of two users divided by the union of the set of items of two users is high, we say there is a high correlation of preferences. The following sets, where S is blue and T is red, are highly correlated:



The following are weakly correlated:



These correlations are defined by the following:

$$A = \{x_p\},\ B = \{y_p\}$$
$$J(A, B) = (A \cap B) / (A \cup B)$$

Item Based Recommenders

Item-based recommendation is all about finding items similar to those that a person (user) already bought. It is not about comparing this person to other similar people (users). The straightforward algorithm is:

```
for each item i that u has no preferences for yet {
      for each item j that user u that has a preference for i {
            compute a similarity s between i and j;
            incorporate u's preference for j weighted by s
                  in a running average;
```

```
            }
        }

        return the top items ranked by weighted average;
```

There are two reasons to choose item-based recommenders over user-based recommenders. The first is that it is more efficient if the number of items is relatively small compared to the number of people. The second reason is that items are less subject to change than users. Similarity of items does not change significantly over time whereas user tastes may vary often. In fact, since item similarities do not change often, they are good candidates for pre-computation to enable the recommender algorithm run faster.

In Mahout, the following illustrates the differences between implementations of user-based vs. item-based recommenders.

A user-based recommender is coded as:

```
public Recommender buildRecommender(DataModel model) throws
TasteException {
   UserSimilarity similarity = new
PearsonCorrelationSimilarity(model);
   UserNeighborhood =
         new NearestNUserNeighborhood(2, similarity, model);
   return new GenericUserBasedRecommender(model, neighborhood,
                                           similarity);
}
```

An item-based recommender is coded as:

```
public Recommender buildRecommender(DataModel model) throws
TasteException {
   ItemSimilarity similarity = new
PearsonCorrelationSimilarity(model);
   UserNeighborhood =
         new NearestNUserNeighborhood(2, similarity, model);
   return new GenericItemBasedRecommender(model, neighborhood,
                                           similarity);
}
```

The next section provides an example of one of the simplest item-based recommender algorithms.

Slope-One Recommender

The concept of this algorithm is based on differences of ratings of an item and ratings of other items. For example, if most people give Trek bikes ratings of 4 stars and Cannondale bikes ratings of 3 stars, which is 1 less, then if you rate Trek bikes with 3 stars, you will likely rate Cannondale bikes with 2 stars, which is one less.

The algorithm is in two parts – finding differences among general ratings and recommending items to individuals.

```
// Calculate differences

for each item i {
     for each item j {
      for each user u expressing a preference for both {
           add the differences in u's preferences to a
                 running avg.;
      }
     }
}

// Recommendation

for each item i that user u has no preference for {
     for each item j that u has a preference for {
      find the avg. preference difference between j and i;
      add the difference to u's preference value for j;
      add this to a running average;
     }
}
return top items ranked by these preferences;
```

As an example, let's say that John, Mark, and Lucy are customers (users) and that there are items A, B and C for them to buy. The buy matrix in this example is:

Customer	Item A	Item B	Item C
John	5	3	2
Mark	3	4	No rating
Lucy	No rating	2	5

What value shall we predict for Lucy's rating of Item A?

Let's first compare items A and B.

$$Lucy_A - Lucy_B = ((John_A - John_B) + (Mark_A - Mark_B)) / 2$$
$$= ((5-3) + (3-4))/2 = (2-1)/2 = .5$$
$$Lucy_A = 2.5$$

Next, let's compare items A and C. Since Mark does not have a rating for item C, we only compare John's ratings.

$$Lucy_A - Lucy_C = John_A - John_C = 5-2 = 3$$
$$Lucy_A = 8$$

Now let's take the average to predict Lucy's rating.

$$Lucy_A = ((2.5 \text{ X } 2 \text{ items}) + (8 \text{ X } 1 \text{ item})) / (2+1)$$

= (5 + 8)/3 = 13/3 = 4.33

Ultimately, based on average differences between items A and B as well as between items A and C, it seems that 4.33 is a reasonable prediction of Lucy's rating for Item A.

Recommenders and Hadoop

The trivial example above can easily be calculated in memory. However, what happens when the data gets too large? In this case, the algorithm cannot run in memory because memory cannot possibly hold all of the data. For example, if the data exceeds 2GB, we would need more than 2.5GB of heap space. This exceeds the maximum limit of heap space that can be allocated to JVMs (Java Virtual Machines).

One algorithmic option for big data could be a sophisticated algorithm on one machine. The algorithm would include lots of member swapping from disk to RAM and back. Essentially, memory swapping moves subsets of data in to memory so that algorithms can perform partial operations and moves partial results back to memory to make room for the next operation on the data. This would continue until all partial results are completed. Then all partial results are brought back into member and aggregated for the final result.

Other options include big data platforms such as Hadoop and Accumulo. Even other options include MPP (Massively Parallel Processing), which is not covered in this course.

Mahout uses Hadoop. Let us explore this option.

Mahout's Item-Based Recommender for Hadoop

This algorithm is conceptually described below. It is a bit complex because it requires multiple MapReduce calls. The code is available in the Mahout book or can be accessed by downloading it from the Mahout site at Apache.

Essentially, the algorithm creates a co-occurrence matrix that represents how many users purchased products X and Y together for all product pairs. Then, the algorithm looks at what some user, say User3, bought and determine what to recommend. It follows the steps of the Slope One algorithm described above. It just goes about the steps for solving this problem differently because the algorithm must be written using MapReduce code.

Suppose we have 7 items numbered 101, 102, …, 107. Let's create a 7 x 7 matrix to represent the co-purchases. To calculate the co-occurrences, the following algorithms is run:

```
for each user u {
   for each item i1 purchased by u {
         for each item i2 purchased by u {
               M[i1, i2] += 1;
         }
   }
}
```

The resulting matrix is:

	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

The diagonal represents that the matrix is symmetric. In other words, the co-purchases between items X and Y are identical to the co-purchases between Y and X (which is obvious).

To find a preference for a specific user, look at the stated preferences for that user. Then look at the co-occurrences of item 101 and all other items purchased (first row). Weight the co-occurrences of 101 per the user's purchases. This gives a weighting to the users probable pension for item 101. Continue to do the same thing for the rest of the items. The calculation for the Recommendation (R) value for the first row is:

	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		38.0
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

(5 x 2.0) + (3 x 0.0) + (4 x 0.0) + (4 x 4.0) +
(2 x 4.5) + (2 x 0.0) + (1 x 5.0)
= 40.0

The calculation for the second row is:

	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

X

U3
2.0
0.0
0.0
4.0
4.5
0.0
5.0

=

R
40.0
18.5
24.5
38.0
26.0
16.5
15.5

(3 x 2.0) + (3 x 0.0) + (3 x 0.0) + (2 x 4.0) +
(1 x 4.5) + (1 x 0.0) + (0 x 5.0)
= 18.5

The subsequent rows are calculated the same way. User 3's non-purchases are where U3's values are 0.0, which are items 102, 103 and 106.

	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

X

U3
2.0
0.0
0.0
4.0
4.5
0.0
5.0

=

R
40.0
18.5
24.5 ⬅
38.0
26.0
16.5
15.5

Since the highest R value is item 103, user 3 is recommended item 103.

The corresponding multi-step MapReduce algorithm is now presented. It is simplified here to make it more understandable.

Step 1: Create the user vectors

Initially, the key-value pairs represent keys as positions in the file and the values are strings that map users to the items they purchased. For instance, this may be **<239, "98955: 590 22 9059">** where **239** is the position in the file and the string **"98955: 590 22 9059"** represents user **98955** and items **590, 22,** and **9059**.

The first Map creates intermediate key value pairs that are of the form user:item.

98955:590 // user:item
98955:22
98955:9059

The Reduce step creates <key, [array values]> for each user.

98955:[590:1.0, 22:1.0, 9059:1.0]

These key-value pairs are used as input to the second MapReduce step.

Step 2: Calculate the co-occurrence matrix

The Map operates on the key-value pairs produced by the last step.

For user **98955:**

Map (98955, [590:1.0, 22:1.0, 9059:1.0])

Intermediate Results – each set of paired items

590:590	***22:590***	***9059:590***
590:22	***22:22***	***9059:22***
590:9059	***22:9059***	***9059:9059***

For user **12345:**

Map (12345, [590:1.0, 67:1.0, 22:1.0]) // some other User Vector

Intermediate Results – each set of paired items

590:590	***67:590***	***22:590***
590:22	***67:22***	***22:22***
590:67	***67:67***	***22:67***

Send these to Reducers that will create the co-occurrence matrix.

Reduce 1

Input:

590:590, 590:22, 590:9059, 590:590, 590:22, 590:67

Output:

590: [22:2.0, 67:1.0, 590:2.0, 9059:1.0]

Reduce 2

Input:

22:590, 22:22, 22:9059, 22:590, 22:22, 22:67

Output:

22: [22:2.0, 67:1.0, 590:2.0, 9059:1.0]

Reduce for items **67** and **9059** are similar.

Step 3: Perform all the Recommendation calculations. Since the co-occurrence matrix is symmetric, we can multiply each column by the user vector instead of each row. In addition, we save much computation time by only performing matrix multiplication when the User has a value that is not 0.

First matrix multiply:

	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

X

U3
2.0
0.0
0.0
4.0
4.5
0.0
5.0

=

R
5x2.0 = 10.0
3x2.0 = 6.0
4x2.0 = 8.0
4x2.0 = 8.0
2x2.0 = 4.0
2x2.0 = 4.0
1x2.0 = 2.0

Second matrix multiply (note that this step is skipped in the algorithm because no new information is calculated):

	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

X

U3
2.0
0.0
0.0
4.0
4.5
0.0
5.0

=

R
10.0 + 3x0.0 = 10.0
6.0 + 3x0.0 = 6.0
8.0 + 3x0.0 = 8.0
8.0 + 2x0.0 = 8.0
4.0 + 1x0.0 = 4.0
4.0 + 1x0.0 = 4.0
2.0 + 0x0.0 = 2.0

Third matrix multiply (note that this step is skipped in the algorithm because no new information is calculated):

	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

X

U3
2.0
0.0
0.0
4.0
4.5
0.0
5.0

=

R
10.0 + 4x0.0 = 10.0
6.0 + 3x0.0 = 6.0
8.0 + 4x0.0 = 8.0
8.0 + 3x0.0 = 8.0
4.0 + 1x0.0 = 4.0
4.0 + 2x0.0 = 4.0
2.0 + 0x0.0 = 2.0

Fourth matrix multiply:

	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

X

U3
2.0
0.0
0.0
4.0
4.5
0.0
5.0

=

R
10.0 + 4x4.0 = 26.0
6.0 + 2x4.0 = 14.0
8.0 + 3x4.0 = 20.0
8.0 + 4x4.0 = 24.0
4.0 + 2x4.0 = 12.0
4.0 + 2x4.0 = 12.0
2.0 + 1x4.0 = 6.0

Fifth matrix multiply:

	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

X

U3
2.0
0.0
0.0
4.0
4.5
0.0
5.0

=

R
26.0 + 2x4.5 = 35.0
14.0 + 1x4.5 = 18.5
20.0 + 1x4.5 = 24.5
24.0 + 2x4.5 = 33.0
12.0 + 2x4.5 = 21.0
12.0 + 1x4.5 = 16.5
6.0 + 1x4.5 = 10.5

Sixth matrix multiply (note that this step is skipped in the algorithm because no new information is calculated):

	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

X

U3
2.0
0.0
0.0
4.0
4.5
0.0
5.0

=

R
35.0 + 2x0.0 = 35.0
18.5 + 1x0.0 = 18.5
24.5 + 2x0.0 = 24.5
33.0 + 2x0.0 = 33.0
21.0 + 1x0.0 = 21.0
16.5 + 2x0.0 = 16.5
10.5 + 0x0.0 = 10.5

Seventh matrix multiply:

	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**

X

U3
2.0
0.0
0.0
4.0
4.5
0.0
5.0

=

R
35.0 + 1x5.0 = 40.0
18.5 + 0x5.0 = 18.5
24.5 + 0x5.0 = 24.5
33.0 + 1x5.0 = 38.0
21.0 + 1x5.0 = 26.0
16.5 + 0x5.0 = 16.5
10.5 + 1x5.0 = 15.5

Step 4: Make the recommendation

Through a series of Maps and Reduces, we produce a vector like this for User 3:

U3: [R: [101:40.0, 102:18.5, 103:24.5, 104:38.0, 105:26.0, 106:16.5, 107:15.5]]

Since we know that User 3 has not purchased 102, 103 and 107, we look at the highest value. This is item 103 for a recommender value of 24.5! The machine recommends item 103 to User 3.

Conclusion

In this module, we discussed multiple approaches to recommending items of interest to customers. These algorithms are called *collaborative filtering* algorithms. The first flavor is *user-based* in which the behavior of the user to whom you are making a recommendation is compared to like users. For efficiency, the algorithm finds the top N users like the current user. Then the algorithm recommends items that other users like the current user liked, but the current users has not yet purchased them. The second flavor is *item-based* in which items similar to items the customer has already purchased are recommended. Mahout provides collaborative algorithms for small data sets and also provides Hadoop MapReduce algorithms for extremely large dataset.