The MultiLens Cookbook

Brad Miller
bmiller@cs.umn.edu

February 1, 2003

Contents

1 Preliminaries

This is a basic little cookbook to get you started using the MultiLens recommendation engine.

You need to have a copy of multilens.jar in your CLASSPATH. You can get a copy from /project/Grouplens/java/jar/multilens.jar. If you want a copy of the source feel free to check one out from cvs. Everything you need should be in the org.grouplens.multilens package, so just import the package with an **import** org.grouplens.multilens.∗; and you should be good to go.

There is also a jre.servlet package in case you want to make your own servlet. If you are really adventurous check out the source code for the jre.soap package and see how to make a soap client.

You also need to have mm.mysql-2.0.11-bin.jar in your CLASSPATH if you are going to use the database functionality.

You need to configure your MySQL database paths. Throughout the rest of this document I will assume the use of the default database information. The values compiled into the jar file will not work for you, but you can set the defaults at the beginning of your program with the following calls:

```
DBConnection.setDefaultURL(”jdbc:mysql://hugo.cs.umn.edu/model”);
DBConnection.setDefaultUser(”bmiller”);
DBConnection.setDefaultPW(”foobar”);
```

It is possible that you may have the model and the user ratings data in two different databases. If this is the case for you, then you can use the following calls to tell the User class where to find ratings.

```
User.setUserDBURL( ”jdbc:mysql://hugo.cs.umn.edu/ML3”);
User.setUserDBuser(”bmiller”);
User.setUserDBpw =(”foobar”);
User.setMyRatingTable(”ratings”);
```

The URL for the database name for DBConnection specifies that this is a mysql database on a machine called hugo.cs.umn.edu using a database called model.

2 Cooking a model

The following Java code builds a new item item model and writes the result to the database.

```
1   public static void main(String args[]) {
2       CosineModel aModel = new CosineModel(500,500);
3       ZScore myFilt = new ZScore();
4       aModel.build(true,”rating”,myFilt);
5       aModel.buildFinal();
6       aModel.setModelTable(”cooked_model”);
7       aModel.writeModel();
8       }
```

The first line simply creates a new model. As indicated by the name this model represents the similarity between two items using cosine similarity. I have also implemented the class **JustCountModel** to represent similarity by simple cooccurrence. JustCountModel has the same interface.

Line 3 creates an object conforming to the ItemVectorModifier interface. I provide two classes that implement this interface **ZScore** and **Normalize**. The ZScore class modifies a **Vector** of **Item** by calculating the average rating and then subtracting the average from each item. Similarly, Normalize, normalizes a vector of items.

Line 4 invokes the model build function. As you can see there are three parameters. If the first parameter is set to true, the build function will ignore ratings less than three. The second parameter specifies the database table to read ratings from. the table can be named anything you want, but it must be defined with the columns shown in the listing below.

```
CREATE TABLE rating (
   gl_id int(11) DEFAULT '0' NOT NULL,
   mlid bigint(20) DEFAULT '0' NOT NULL,
   rating float DEFAULT '0' NOT NULL,
  UNIQUE u_i_pair(gl_id ,mlid)
 );
```

The call to buildFinal on line 5 finalizes the model and prepares it to be written to the database. Once you have called buildFinal the model can be used for making recommendations and predicitons even if you have not saved it to the database. The buildFinal call is unnecessary if you are using a **JustCountModel**.

The call to setModelTable on line 6 tells the model that it will be stored in a table called "cooked_model". The table can be called whatever you like, but it must have the columns and index as specified by the following:

```
CREATE TABLE cooked_model (
   item_row bigint(20) DEFAULT '0' NOT NULL,
   item_col bigint(20) DEFAULT '0' NOT NULL,
   sim_score int(11) DEFAULT '0' NOT NULL,
  UNIQUE i_i_pair(item_row ,item_col)
 );
```

Finally, the cooked model is written to the database by the call to writeModel on line 7. That's it, thats the whole process to build a model and store it to the database.

3 Making Predictions

If you want to use a model to make some predictions, the following bit of code shows you how it is done:

```
1 CosineRecModel aModel = new CosineRecModel(1000,1000);
2 CosineRec myRE;
3 aModel.setModelTable("normed_model");
4 aModel.readModel(1000,false);
5 myRE = new CosineRec(aModel);
6 User currentUser = User.getInstance("12345");
7 Prediction p = myRE.ipredict(currentUser,5463);
8 double pred = p.getPred();
9 p.print();
```

Lets take this apart. Line's 1–3 should be familiar to you if you read section 2. As a reminder, we are declaring a model and a recommender and telling the model that it is going to get its data from the table called "normed_model".

The call to readModel on line 4 instantiates a new model by reading information from the database. The first parameter tells the model to truncate itself by only reading the 1000 most similar items in each rown of the matrix. The second parameter tells the model not to instantiate the lower triangle of the matrix. Since the model is symmetric this can save a lot of space. I'll talk about when you might want to set this parameter to **true** in section 6.

Depending on the size of your cooked model and what you set the first parameter to, the read process can take several minutes. Now that you have a model you can create a recommender that uses the model, which is illustrated on line 5.

Line 6 creates a new **User** object. Creating a user this way has the side effect of reading in this users ratings from the database.

Finally on line 7, we actually generate a prediction. Notice that ipredict takes a User and an item identifier as its parameters. ipredict returns a Prediction object that that encapsulates a bunch of stuff about the prediction including the predicted rating which you can access with getPred(). I like this method because it gives us good flexibility down the road for adding new stuff to prediction (confidence, etc.). But if you are really super concerned about memory and not into creating objects you can call predict(User,int) which returns a double.

4 Making Recommendations

This next recipe illustrates how to create a recommendation list.

```
CosineRecModel aModel = new CosineRecModel(1000,1000);
CosineRec myRE;
aModel.setModelTable("normed_model");
aModel.readModel(1000,false);
myRE = new CosineRec(aModel);
User currentUser = new User("12345");
TreeSet myRecs = getRecs(currentUser, 10, false, null);

Iterator it = myRecs.iterator();
while (it.hasNext()) {
    Recommendation rec = (Recommendation)it.next();
    System.out.println("Recommended item = " + rec.getItemID());
}
```

Notice that the setup is just the same as in section 3 in terms of creating the model objects and the User object.

The call to getRecs takes four parameters. A user object that contains all of a users ratings.

The second parameter is the number of items for getRecs to return as part of the recommendation list. If you want all possible values returned you can simply pass a value of zero to getRecs for this parameter.

The third parameter is a flag that if **true** will include a users previously rated items as part of the recommendation list. The normal value for the flag is **false** so that only new items are returned as part of the recommendation list.

The fourth parameter (which as you can see may be null) is a HashSet of item identifiers. I call this a 'filter set'. The effect of passing a filterSet to the recommender is that only items in the filterSet will be considered as part of the recommendation process.

For example in the servlet implementation of the recommendation engine I create several filtersets at startup that contain the item ids of the movies that are in each genre. Now if a user asks for Science Fiction recommendations I can pass in the filterSet for Science Fiction and save a lot of time.

Arbitrary filtersets can be constructed quite easily from a database query.

Finally, notice that getRecs() returns a TreeSet. The returned TreeSet contains Recommendation objects. The nice thing about a TreeSet returned is that you can iterate over it and get the Recommendations back in order from most similar to least similar. This iteration process is illustrated by the last four lines of the listing above.

5 Predictions for made up Users

If you want to make up a user and have that made up user rate a set of items in a certain way and then get predictions for that user you can use the following trick.

```
User currentUser = new User();
currentUser.addRating(1234,5);
currentUser.addRating(5678,2);
currentUser.addRating(4321,4);
Prediction p = myRE.ipredict(currentUser,5463);
double pred = p.getPred();
p.print();
```

You can use this trick to make a blank user and then have the user any set of items any way you want.

6 Shopping Cart Recommendations

What if you want to get recommendations, but not necessarily for a particular user. Maybe you want to recommend some items that go with the items a user has placed in a shopping cart. Maybe you want to recommend some other items that are similar to items that a user has just been looking at. Here are some other questions that might point you towards this recipe:

- What other items go with (are similar to) this list of items.
- Show me more items like these.
- I want to pick out a gift for my nephew and I know he likes these

The way you get recommendations in this way is almost identical to the previous recipe. The difference is shown in the listing below where on line 9 we pass a Set Object called marketBasket to getRecs rather than a User Object.

```
1 CosineRecModel aModel = new CosineRecModel(1000,1000);
2 CosineRec myRE;
```

```java
aModel.setModelTable("normed_model");
aModel.readModel(1000,true);
myRE = new CosineRec(aModel);
User currentUser = new User("12345");
Set marketbasket = new Set();
marketBasket.add(new Integer(1234));
TreeSet likeR = myRE.getRecs(marketBasket,10,false,null);

Iterator it = myRecs.iterator();
while (it.hasNext()) {
    Recommendation rec = (Recommendation)it.next();
    System.out.println("Recommended␣item␣=␣" + rec.getItemID());
}
```

Another important difference in this example is that you will notice that the flag to readModel that indicates we should instantiate the lower triangle is set to true. The reason for this goes all the way back to the way that models are built.

7 Combining Models

What if you want to combine one or more similarity models together to improve the recommendations? For example, you have ratings of restaurants along multiple dimensions including 'food quality', 'service', and ambience. Another exmample would be to include content based similarity with quality based similarity. For example I have built four models for movie recommendations:

- A *quality* model based on user rating of movies
- A *cast* model based on the cooccurrence of famous cast members and directors in the movie.
- A *genre* model based on what genre or genres a movie is in.
- A *plot summary* model based on TFIDF analysis of the movie's plot summary.

These models can be combined in arbitrary ways using the functions shown in the listing below:

```java
1 AddCell myAdder = new AddCell();
2 aModel.setModelTable("cooked_model");
3 aModel.readModel(1000,false);
4 aModel.combineModel(1000,false,"cast_model",myAdder);
```

The call that actually does the combination is on line 4. The call to combineModel takes a model from the database and incorporates it into an existing model. The first two parameters are the same as to readModel indicating truncation, and whether to instantiate the lower triangle of the matrix.

So far, I have not seen the need to provide an operator that takes two models as parameters and creates a third new model as a result. Please let me know if you see a need for this in your application and it would be easy to add.

The third parameter is the name of a database table that contains the model you are combining with the first. In this case we are combining the cast model with the quality model. With a second call to combineModel we could add yet a third model.

The fourth parameter contains an object that implements the CellCombiner interface. As you may guess the myAdder object combines two cells of the matrix by adding them together. Another simple CellCombiner is included in multilens jar file called MultCell this multiplies the contents of two matrix cells together. If you want to combine cells in more complicated ways, maybe by weighting one matrix more heavily than another, you can do it by implementing your own CellCombiner.

If you have two models already instantiated (maybe by building them from files) and you want to combine them, by incorporating one into another you use the following:

```
        CosineModel aModel = new CosineModel(1000,500);
        CosineModel bModel = new CosineModel(1000,500);
        AddCell myAdder = new AddCell();
        ZScore myFilt = new ZScore();
        aModel.buildFromFile("/Users/bmiller/Projects/ml-data/u2.base",myFilt
        aModel.buildFinal();
        bModel.buildFromFile("/Users/bmiller/Projects/ml-data/u3.base",myFilt
        bModel.buildFinal();
        aModel.combineModel(bModel,myAdder);
```

the results of the above operations are that the similarity scores in bModel are added to the similarity scores in aModel. The result is that aModel is changed, but bModel remains unchanged.

8 Filtering on Broad Categories of Items

We covered this topic briefly in section 4. But I'll highlight it again here. The idea is that a user would like recommendations from some broad class of objects. Note that this is not as well specified as I would like to know which the these n specific objects I will like best. The HashSet is a way to specifiy the broad set of objects and allow the recommender to efficiently filter out items that don't meet the criteria before doing much calculation.

I use this feature heavily in the servlet version of the recommender system to make movie recommendations for specific generes of movies.

```
HashSet myGenre;
TreeSet likeR = myRE.getRecs(marketBasket,10,false,myGenre);
```

9 Avoiding the Databse (Cooking with files)

So you want to use files, rather than that nice spiffy database to manage all of your ratings and test cases. Ok, here's how you do it:

```
        CosineModel aModel = new CosineModel(1000,500);
        AddCell myAdder = new AddCell();
        ZScore myFilt = new ZScore();
        aModel.buildFromFile("/Users/bmiller/Projects/ml-data/u2.base",myFilt
        aModel.buildFinal();
```

Once you have a model you can use it just like any other model per the instructions above.

File Format

The file format that buildFromFile expects is a tab delimeted file, with one row per record of the form userid itemid rating. user and item identifiers are parsed as integers. The rating field is parsed as a double, and so it will accept either whole number ratings or half star ratings. Actually it will accept any real number as a rating right now so we could use quarter or eighth stars if we wanted to.

The standard set of MovieLens test data (ml-data.tar.gz) will work right out of the box. However, it is good to remember that the buildFromFile function expects that the file is sorted on userId (the first column of numbers). If the file is unsorted you can get very strange results, and your build time will be much slower.

The buildFromFile method saves the ratings for every user in an internal cache, this eliminates the need for a rating database. To get a User instance, you should use the new static factory method, User.getInstance("userid");. The buildFromFile method uses the User.addUserToCache(User); function to create the cached users.

You can also use the cache functionality when retriving users from the database. The User.getInstance function is smart enough to look in the cache first and then go to the database if appropriate. If you are running offline tests and know that the test users will not be adding new ratings to the database, using the cache can speed up your tests significantly.

10 Adding Your Own Recommender

So you want to make your own recommender. You have one big decision to make first: Can you use an existing similarity model or will you need a new similarity model? MultiLens currently supports a model based on Cosine similarity and one on simple item cooccurrence counts. If you want a different model you will need to implement a new class. Your starting point is the GenericModel class which gives you some utility functions, access to all the Sparse Matrix functionality, and defines some functions that you must implement.

If you can use an existing model then you just need to develop a recommender class. The recommender class must implement a constructor that takes a model as a parameter and implement the getRecs and ipredict methods.

Lets walk through a specific example. Suppose that you are an evil marketing person (emp) who wants to force certain recommendations on unsuspecting customers, based on the contents of their shopping cart.

Rules can be represented as a matrix! Rows are the antecedants and columns are the consequents.

Model cooking 'boils down' down building an interface that allows the emp to express rules in a more natural if then like language and stores them in a table. Figure x shows a simple example of this idea.

To implement the model that we'll use to build the model for this we will create a new subclass of Generic model, lets call this class RuleModel.

In addition, we'll need to define a subclass of SparseCell to represent what gets stored in the matrix.

For example the SparseCell used to build a CosineModel is defined as follows:

```java
public class CosineModelCell extends SparseCell {
  private double partialDot;
  private double lenI;
  private double lenJ;
  private int count;
  // accessor functions after this...
}
```

For our RuleBased model we might decide to represent our Sparse Cell as a single boolean value as follows:

```java
public class CosineModelCell extends SparseCell {
  private boolean isOn;
  // accessor functions after this...
}
```

GenericModel itself extends SparseModel, so right away we get all of the sparse matrix handling for no extra charge. Every subclass of Generic Model must implement the following methods:

int getDBRepSimScore(**int** i, **int** j) Returns the integer representation of the similarity score to store in the DB. CosineModel implements this method by getting the double val and multiplying by 1000. **void** setDBRepSimScore(**int** i, **int** j, **int** val) Take the integer similarity value, represented as an integer in the database, convert it to the appropriate internal value, and store it in the appropriate cell of the matrix. In our RuleModel case we may simply check to see if the integer value is greater than zero. If so, then store true.

void addDBRepSimScore(**int** i, **int** j, **int** val, CellCombiner myOp) Take the integer similarity value, represented as an integer in the database, convert it to the appropriate internal value, and combine it using myOp with any value already present at i,j and then store the result.

void insertUserItems(Vector userItems) Take a vector of Items, process them according to how you want to build the model, and store the result. For example the CosineModel version of this function updates a bunch of partial dot products and vector lengths.

Each subclass of GenericModel should also provide accessor functions to get access to the contents of the Cell. (Its possible that we might want to refactor this design so that GenericModel provides functions that just return SparseCells and put the appropriate functionality in the subclasses of SparseCell – it would be a fair amount of work)

The good news for our example is that we can use the same model for both building and recommending. In the CosineModel, I chose to make a separate subclass for building and recommending because the recommender did not need to store all the partial values, and I wanted to save memory.

So, the next step is to implemente our recommender class. Lets call it RuleRec. RuleRec should implement ipredict and getRecs. ipredict should just throw an exception saying that it is not appropriate. get Recs should work as follows:

- For each item the user has rated check the row of the matrix. Add any itemIds to our candidate list for which the cell is true.
- Return all the items on the candidate list.