Assignment 2

DSCI 641 - Recommender Systems – Winter 2024

OVERVIEW

In this assignment, you will use recommender systems data from Assignment 1 to train, evaluate, and explore several recommendation algorithms. Some of these come from LensKit; two of them you will implement.

The deliverables for this assignment are:

- A **zip file** of your project (scripts, notebooks, etc – everything I would need to re-run it, except the data files)
- A **PDF export** of your Jupyter notebook with the results of your evaluation and exploration.

This assignment is due **11:59 PM** on **Saturday, February 10**. Some of the computations will take quite some time to run (e.g. recommending for all users in MovieLens 25M with item-item CF takes 8-12 CPU-hours), so I recommend starting early. I have provided very rough attempted time estimates in this assignment; I welcome feedback on their accuracy.

Revision Log

Jan 31	Removed the BPR algorithm.
	Specified that ML-20M is acceptable.

DATA AND SOFTWARE

This assignment will use the **MovieLens 25M** data set that you used for Assignment 1. I recommend using the **Latest Small** data set to test your code, to make sure the whole evaluation is working, and then using 25M to get final results. If your compute resources do not permit a full cross-fold evaluation on 25M, you can evaluate instead with 5 disjoint 5000-user splits (use `sample_users` instead of `partition_users`). You can also use the 20M data set instead and receive full credit.k

In addition to the basic PyData stack, this assignment will use LensKit. The Conda environment file attached will set up an environment with all necessary software; you can also install LensKit with `pip`.

The Getting Started guide is useful for learning how to train a recommendation algorithm with LensKit and compute an evaluation metric.

THE ASSIGNMENT

There are several components to this assignment, described below. I will also discuss some how-to in class on Feb. 6, and we will start working with examples.

Preparing Data

Use LensKit's data splitter to partition the MovieLens data set for 5-fold cross-validation. The example code there works: partition users, and select 5 ratings/user for testing. Save the partitions into files (Parquet files are good for efficiently re-loading the data). The LensKit MovieLens class can help you efficiently load data (ML-25M, 20M, or Latest). Similar classes also exist for other versions of MovieLens.

Note that LensKit requires columns to be called 'user', 'item', and 'rating'; if you use the MovieLens loader, it will automatically rename the columns for you.

Tip: you can work with the "Latest Small" data set to debug your code and experiments. It has the same format as ML-25M, but is smaller.

Write a script or notebook to split the data and save the results. Working in a notebook can be useful to combine splitting and a descriptive analysis of the data set, but needs more work to integrate into an automated workflow.

I estimate this part of the assignment to take approximately **1 hour**, hopefully less.

Evaluating Recommendation Algorithms

Use the MovieLens 25M data set to train and evaluate several collaborative filters. For each, you will measure the top-N **recommendation** accuracy using nDCG, and for rating predictors, the rating **prediction** accuracy using RMSE, on your split data set from the previous section.

ALGORITHM	CONFIGURATION[1]	OUTPUTS
POPULAR	`basic.Popular()`	R
BIAS	`bias.Bias()`	R, P

[1] This refers to the algorithm in its submodule in `lenskit.algorithms`; for example, `als.BiasedMF` is under `lenskit.algorithms.als.BiasedMF`.

USER-USER	`user_knn.UserUser(30, min_nbrs=2)`	R, P
ITEM-ITEM	`item_knn.ItemItem(20, min_nbrs=2)`	R, P
ITEM-ITEM SUM	`item_knn.ItemItem(20, 2, feedback='implicit')`	R
EXPLICIT MF	`als.BiasedMF(50)`	R, P
SVD	`svd.BiasedSVD(25)`	R, P
IMPLICIT MF	`als.ImplicitMF(50)`	R
LIFT	*Provided*[2]	R

For each algorithm, for each partition, do the following:

1. Create an instance of the algorithm object
2. Wrap the instance in a recommender (`Recommender.adapt(algo)`)
3. Fit the algorithm on the training data
4. For a prediction algorithm (P above): generate predictions for all test user-item pairs using `batch.predict`.
5. Generate 50-item recommendation lists for test users with `batch.recommend` (if your test data is in a frame in a variable called `test`, then `test['user'].unique()` will get the test users).
6. Measure the recommendation quality (using `RecListAnalysis`).

I strongly recommend that you write one or more scripts to load the data, train the algorithm, compute recommendations, and write the results. Exactly how you divide the work into different scripts is up to you. My usual solution would be to write a script that takes an algorithm name as a command-line option, loops over the test partitions, and for each partition, trains the algorithm and saves the predictions and recommendations to either CSV or Parquet. In some cases I have two such scripts: one trains the algorithm and saves it to a file with binpickle, and the second loads the pre-trained models and generates recommendations.

[2] This is provided in the file '`lift.py`', which is also an example for you to refer to in writing your own algorithm.

Your scripts need to be *import-protected* (see Scripts and Modules, slide 8) – if they are not, they will mess up LensKit's parallel processing. Also, if you turn on logging, you will get useful LensKit messages:

```
import logging
# lots of stuff...

if __name__ == '__main__':
    logging.basicConfig(level=logging.INFO)
    do_my_stuff()
```

I then write a notebook that loads the recommendations into a Pandas data frame, and uses `RecListAnalysis` to compute the metrics and display the results. Note that `RecListAnalysis` is only for computing top-N metrics like nDCG; rating prediction metrics, like RMSE, you can do directly on predictions data frame (and if you pass the whole test data frame to '`predict`' to generate it, it will include the test ratings in the returned prediction frame).

At the end of this, you should have two bar charts or dot plots: one showing RMSE for each rating prediction algorithm, and another showing mean nDCG for each of the algorithms. You may also use a box plot, showing the individual user nDCGs and RMSEs (if you compute RMSE grouped by user).

What algorithm(s) seem to perform the best for each task?

I estimate this part of the assignment to take approximately **4 hours** of your work, and up to another **20 hours** of compute time. Once you have things figured out for 1-2 algorithms, the rest should be doing the same thing with additional algorithm definitions.

Content-Based Recommendation

Implement a new recommendation algorithm that uses the MovieLens Tag Genome to recommend movies that are similar to a user's rated movies. The Tag Genome comes with the ML-25M data set, in the `genome-scores.csv` file. This file contains three columns: movie ID, tag ID, and relevance (the meaning of the tag IDs is in `genome-tags.csv`). It is complete, in that it has relevance scores for every movie for a large number of tags, although it does not cover every movie in the MovieLens data. The logic is a variant of what you used in A1, except set up to work for many users.

You can think of this file as defining a vector for every movie: an approximately 1500-dimensional vector describing the movie in terms of Tag Genome tags.

Write an algorithm that computes scores a movie by:

1. Computing a user tag vector by taking the average of the tag vectors for the movies the user has watched, weighted by their rating (that is, the user's value for a tag is the average of their movies' value for that tag. You can do this relatively efficiently with NumPy vectorized operations.). Note that this is *different* than the item neighborhood logic in A1.
2. Compute the Pearson correlation between the user tag vector and the movie's tag vector.

To do this, you need to write a Python class that extends the Predictor base class, and implement the following methods:

```
class GenomeRec(Predictor):
    def fit(self, ratings, genome):
        # do what you need to do to prepare your data
        # maybe convert genome into a matrix?
        # maybe normalize it movie vectors are unit vectors?
        # save in a field ratings so you have them later for users
        #
        # this method needs to set up anything prediction will
        # need from the training data.

    def predict_for_user(self, user, items, ratings=None):
        # compute scores for items, for user
        # the return value needs to be a Pandas series
        # whose index is 'items'
        # unscorable items (e.g. not in the genome) can have
        # score np.nan
        # ratings will usually not be provided here
```

Look at my Lift recommender (in the attached file `lift.py`) for an example of a complete, working recommender without too many bells and whistles. Add your genome predictor to the algorithms tested in your evaluation in the previous section. Does it make sense to test it as a rating predictor? Only compute RMSE if that makes sense – otherwise just compute nDCG. If you set up your code with scripts to save results, then you should be able to run this and get the results, and re-run the notebook to compute metrics and plots, without re-running the collaborative filters.

You will need to pass the genome data by name once you have wrapped the predictor in a recommender.

```
genome = GenomeRec()
algo = Recommender.adapt(genome)
algo.fit(ratings, genome=genome)
```

The MovieLens data loader can load the tag genome too.

The batch recommendation routines will **not** look up a user's ratings – you need to save the rating data in a field in '`fit`', and then look up a user's historical ratings from that data structure in '`predict_for_user`'. As a matter of convention, any field you set up in fit needs to have its name *end* with an underscore (e.g. `user_ratings_` is a good name for the field that stores users' ratings) – this is usual SciKit-Learn practice.

I estimate this part of the assignment to take **5-10 hours**, depending on how quickly the key ideas come to you. If your recommender is unusably slow, you may want to test on 5 1000-user samples (for a total of 5K users) instead of all 162K users in 5 partitions (use `sample_users` instead of `partition_users`). It isn't ideal, but will let you see the key ideas if your algorithm implementation is too slow.