Tutorial

Anders Geil, Sol Park, Yinuo Jin

Overview

Probabilistic Matrix Factorization (PMF) is a popular class of graphical models commonly used for recommender systems. In this project, we provide a parallel implementation of Gaussian matrix factorization. This tutorial briefly covers the basics of PMF models, then delves into the technical details of how to set up and use our application; first for model fitting (section 3) and then for prediction/recommendation (section 4). Users experienced with PMF models may skip section 1, and go directly to the technical prerequisites in section 2.

§ 1. Probablistic Matrix Factorization Basics

Probabilistic Matrix Factorization models belong to a larger class of collaborative filtering models. As opposed to content-based filtering, these models rely on users' interactions with a fixed set of items to provide recommendations. In a nutshell, PMF models assign to each user a latent (meaning unobserved, inferred from data) vector representing her preferences. Similarly, each item (e.g. a movie) is assigned a latent vector representing its attributes. We expect high ratings to occur for items whose attribute vectors are similar to a user's preference vector. To obtain the latent preference and attribute vectors, however, we need to first *learn* them from data. In this application, we therefore expect the dataset to take on a certain form. We will cover this in the next section.

§ 2 Prerequisites

In this project, we expect the data to come in the form of two .csv files with (at least) 3 columns. The first file contains user-to-item *rating* information. The header line *must* have the column titles "userId", "itemId", and "rating" as the first three columns (in that particular order, left to right). Any columns after that will be ignored. The table below serves as an example of the expected format of a dataset:

userId	itemId	rating
1	1	4.0
1	3	2.5
⋮	⋮	⋮

The second file contains the *supplementary* information for each of the **itemId** records in the first file. Similar to the first .csv file, we expect an exact format for the header in accordance to the example below:

itemId	itemName	itemAttributes
1	Toy Story (1995)	Adventure|Animation|Children
3	Grumpier Old Men (1995)	Comedy|Romance
⋮	⋮	⋮

In this tutorial, we will be using the provided MovieLens dataset containing records of users' ratings on various movies (items). But you may exchange it for any dataset of your choice, as long as it follows the format outlined above.

Before proceeding, however, please verify that the following external dependencies are installed in addition to a working copy of C++17 or higher:

- Eigen (v3.3.9+)
- Boost (v1.73.0+)
- CMake (v3.20+)

§ 3. Getting started

This section will guide you through loading your datasets into the program. To load your ratings dataset, use the option `-i` or `--input` to input its filepath. Similarly, to load your items dataset use `-m` or `--map`. However, if you opt to use the provided MovieLens datasets, simply pass in `--use_defaults` or `-d`.

It is important to briefly note the preprocessing procedure that is applied to the loaded ratings data: The ratings will first be centered to zero, and the rows (observations) will then be shuffled into a random ordering to assure independence between consecutive rows.

In our running example, we will use the provided default datasets.

`./main.tsk -d`

Immediately, you will notice that the program will start to fit the model. The default behavior for the program is to train. However, we also support another mode just for generating recommendations. This option can be set with `--task`, where the options are either `train` or `recommend`. The recommendations option will be covered in more detail later. For now, we will focus on the training mode.

Before moving on to the next section on 'Fitting & tuning the model', there is one more optional argument to highlight. This is the `--output` option which defaults to "results". You may have noticed that a `results` subdirectory was created after you ran the task. This directory will be where the output of the training data will be saved.

§ 4. Fitting & tuning the model

Since we are now more familiar with the fundamental arguments needed to load a dataset, let's try to use it to fit our first model. We will first illustrate how to fit the model sequentially. The default behavior is actually to run the fitting with parallelization. We will first use the sequential mode to go over the various tuning parameters for the model. Then we will go in depth with the more parallelization-specific optimization parameters.

```
./main.tsk -d --run_sequential
```

Notice that the model prints out several different statements as the program is running. Let's briefly discuss what each of them mean.

The first statement you will likely notice is the "epoch" count. In machine learning, we refer to an epoch as a full pass over the dataset, i.e. an epoch ends when the model has seen (and learned from) each entry in the dataset exactly once. We use this to measure how far along the model has come in the fitting procedure. We may adjust the total number of epochs the model is set to process by the command line

parameter `--n_epochs` (set to 200 by default) as follows:

```
./main.tsk -d --run_sequential --n_epochs 400
```

How do we know how long to train the model for? Well, we want to keep fitting the model as long as our latent variable vectors keep updating. Once the updates to our vectors become sufficiently small, we say the model has *converged*. We can assess model convergence by checking that the "loss" starts to stabilize around a narrow range of values, this typically suggests that the model has found a set of optimal parameter values for the latent vectors. Indeed, the loss is exactly the second statement we notice in the printout on our command line.

OK, so our model seems to converge in this case. But what can we do if this doesn't happen as smoothly on another dataset? In that case, there are still a few more parameters we can tune. First, if the loss changes only very slowly, we may want to increase the learning rate to a higher value. We can set this by using the command line argument `--gamma` as follows:

```
./main.tsk -d --run_sequential --gamma 0.1
```

Note that the default value for `gamma` (the learning rate) is 0.01. We recommend adjusting it by a factor of 10 until reaching the desired effect, but beware that setting the learning rate too high may result in the algorithm "jumping out of" the global optimum. For this reason, it is typically better to start with a higher learning rate, and then gradually decreasing it to get a more precise estimate of the latent vectors' parameter values.

Once we start working with smaller learning rates, however, we may also want to get a more granular view of the loss values. For this purpose, we may adjust the frequency by which the loss function is computed using the command line argument `--loss_interval`. That is, how many epochs are we willing to wait between computations of the loss function? For example, the following snippet computes the loss function after *every* epoch:

```
./main.tsk -d --run_sequential --loss_interval 1
```

Beware that this may slow down the model fitting slightly, especially in the sequential version (we'll cover the parallel implementation in a minute). For this reason, we only compute the loss function every 10 epochs by default.

Great, now we know how to adjust the optimization parameters to more effectively fit the model. How, then, can we build confidence that our fitted model has indeed learned a useful representation for our latent preference and attribute vectors? To do this, we measure the *root mean squared error* (RMSE) of our model's predictions on unseen test data. RMSE is a useful measure because it elegantly captures the number of points our model's predictions can be expected to be off by the actual (observed) ratings on the original rating scale.

You may have noticed the program printing out three measurements of the RMSE just before terminating. RMSE(0) represents the expected error of a (naïve) model predicting *only* the middle of the rating scale for every user-item pair. Similarly, RMSE(mean) represents the expected error of a (slightly less naïve) model predicting *only* the average rating across the training dataset. Lastly, RMSE(pred) represents the expected error of our model's predictions using the learned feature vectors. If the error of our trained model is lower than the two other benchmarks, we know the model learned something useful. For the sample MovieLens dataset, we should generally expect to see an error in the range of 0.9x – considerably lower than the benchmarks.

What if the model's RMSE is not better than the benchmark measurements – what can we do to improve the RMSE in general? The best tool in this case is to adjust the *size* of the latent vectors. This can be achieved by using the `--n_components` command line argument as follows:

```
./main.tsk -d --run_sequential --n_components 5
```

Admittedly, adjusting this parameter requires a bit of *fingerspitzgefühl*. On the one hand, making the latent vector size too small may prohibit the model from fully expressing the underlying structures in the dataset. But on the other hand, making the latent vector size too large may give the model too much freedom to orient the vectors in ways that are only representative of the training data, not generalizing well to new (test) data and generating poor recommendations.

Another approach is to adjust the split-ratio between training and test data. Before fitting the model, the program randomly partitions the dataset into a training set (used for model fitting) and a test set (used for model evaluation). If the provided dataset is sufficiently large, it may make sense to dedicate a larger fraction of the dataset to training and a smaller fraction to testing. In this way, the model may learn more of the latent features in the data set, but without compromising the integrity of the evaluation since the test data is still of a reasonable size. The split-ratio between training and test data can be set as follows:

```
./main.tsk -d --run_sequential --ratio 0.8
```

For smaller datasets, we recommend keeping a split-ratio of 0.7 to maintain a stable representation of the model's performance on unseen data. If the split-ratio is set too high compared to the size of the dataset, the RMSE may fluctuate wildly between runs (even using the same hyperparameters) simply because of the random selection of observations to be used in the test data.

Now that we've covered the model tuning parameters, we can finally discuss how to speed up the model fitting with parallelization. As mentioned earlier, by default, the model fitting will run with multiple threads parallelizing the updates to the vectors. So far we have been running in sequential mode with the `--run_sequential` option. We can now remove this argument.

```
./main.tsk -d
```

By default, we use 20 *extra* threads for this parallelization. It is important to note here that when the model is fitting in parallelization mode, it will create 1 thread for the loss computation and reserve the remaining 19 threads for updating the item and user vectors. With this default parameter of 20 threads, the model will split the training data into 19 batches of (roughly) equal-size, disjoint submatrices and compute the parameter updates in parallel.

The number of threads to be used can be adjusted with the `--thread` option. For example, we can increase the extra thread usage to 90, where 1 thread will compute the loss computation, while the remaining 89 threads will fit the latent vectors to the 89 submatrices of the training data in parallel.

```
./main.tsk -d --thread 90
```

Due to the nature of thread allocation in parallel mode – always one for loss computation, and at least one for the fitting – the minimum number of threads that can be specified is 2. Furthermore, if `--run_sequential` is specified, it will always override the parallelization, and ignore the number of extra threads if provided.

§ 5. Using the model for recommendations

After learning the latent user preference and item attribute vectors from the dataset, we're now ready to generate recommendations. The program supports two general methods for generating recommendations. Let's see an example of the first "recommendation" mode and break down the details later.

```
./main.tsk --task recommend -d --user
```

Since the terminal in nature is "stateless" and doesn't store information between separate runs, we need to remind our program about the items in the *rating* dataset and their *supplementary* information. Here, we use the default input directories for dataset loading as previously demonstrated. You'll notice from the terminal that the program is prompting you to `Please specify user id:`. Simply fill in a user id and press return, the program will then display the top 10 recommended items that the user with the chosen id is predicted to like the most.

This is the first type of recommendation we provide: recommending items with attribute vectors most similar to the preference vector of the given user. If a user was not represented in the initial dataset, we have no way of knowing their preferences. In this case, we provide another recommendation scheme: recommending items from other items. Let's try the example below:

```
./main.tsk --task recommend -d --item
```

Similar to before, we simply type an item name and the program will proceed to return the top 10 most "similar" items. Please keep in mind that the PMF model won't necessarily recommend items with similar attributes (e.g. recommending comedies from comedies), but rather items that were generally liked by other users who also happened to enjoy our chosen item.