Tutorial - Parallel Probabilistic Matrix Factorization

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 file 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. 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 all the **itemId** appeared 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
⋮	⋮	⋮

Before proceeding, please check 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

We provide a light-weighted command-line program compiled from our C++ project to apply PMF model on the dataset. This comes in with two steps: (1). we need to fit the PMF model to *learn* the latent features of each user and item; (2). we apply the model learned from the input dataset to perform *recommendations*. This section will guide you through loading your datasets into the program.

First, to instruct the program which task it will perform, use the option `--task` : `--task train` will fit the dataset to learn the PMF model, and `--task test` will use learned model to make recommendations.

To load your *ratings* dataset, use the option `-i` or `--input` to input its file path; Similarly, to load your item's *supplementary* dataset with `-m` or `--map`. However, if you opt to use our provided Movielens datasets, just pass in `--use_defaults` or `-d`.

In our first 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, since our model can't make reasonable recommendations without seeing actual data. However, we also support another mode for recommendations, and our `--task` option provides you the interface to switch between "training" mode (`--task train`) and "recommendation" mode (`--task recommendation`). For now, we will focus on the training mode.

§ 4. Fitting & tuning the model

After familiarizing yourself with the fundamental arguments you need to read in datasets, let's try using it to fit our first model. The program's default behavior is to run the model fitting parallelization, as it's the most exciting feature we offer. But let's illustrate fitting the model sequentially first. For this section, we will go over various tuning parameters specifically related to the PMF model with sequential mode, and then we will go in depth in optimization with parallelization.

```
./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 means.

The first statement you'll likely notice is the "eposh" 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 is 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*: the "loss" starts to stabilize around a narrow range of values. It typically suggests the model has found the optimum parameter values for the latent vectors. Indeed, the "loss" is exactly the second statement we notice in the printout on our command line.

OK, you might see that our model converges in the provided example, but this doesn't always happen smoothly on another dataset. To deal that case, we'll introduce a few more parameters we could tune. First, if the loss changes slowly, we may want to increase the *learning rate*, simply by setting the `gamma` parameter in our program:

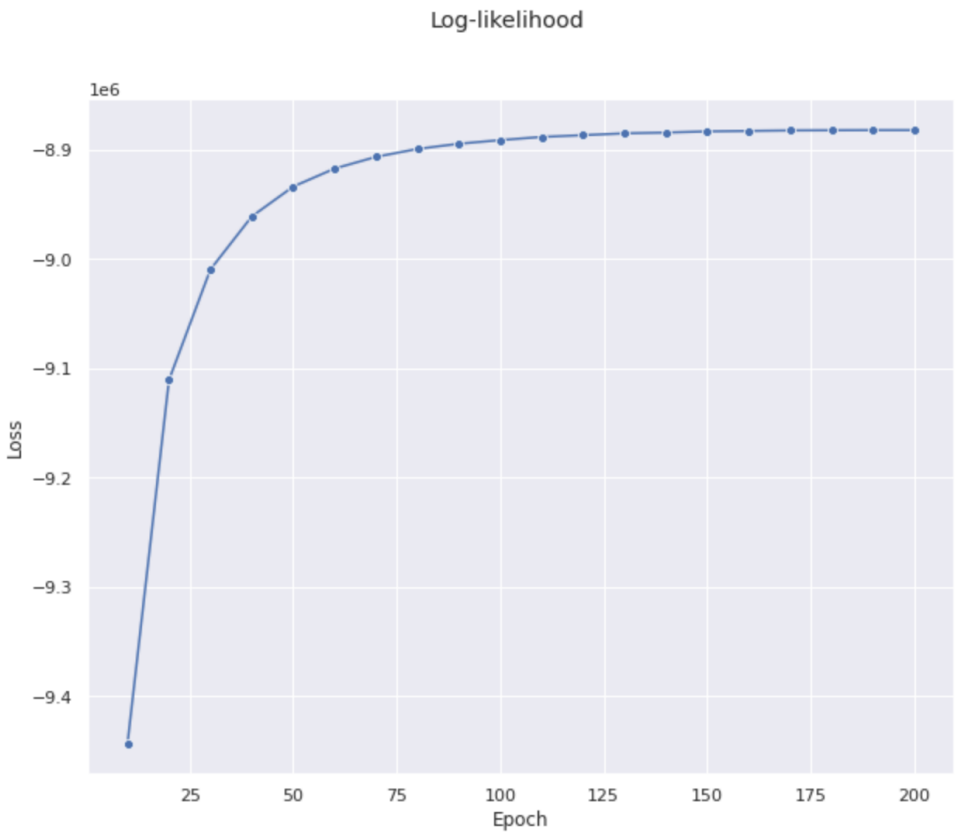
```
./main.tsk -d --run_sequential --gamma 0.1
```

The default `gamma` value is set to 0.01, and we recommend adjusting it by a factor of 10 until reaching the desired effect. In general, a higher learning rate may lead the model to jump "out of" the global optimum and impair the optimization, whereas a lower learning rate results in more fine-grained loss update. For this reason, it's typically better to start with a relative high `gamma` and gradually decrease it to have a more precise estimate of the latent vectors' parameters.

To see how our model works in progress, we refer to the *loss function values*, which in general measure how likely we observe the dataset given the model parameter values at each step ($\log \Pr(\mathrm{Data}, \mathrm{Model})$) . Then how often we want to compute such loss function? We provide an option to adjust that:

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

It specifies the program to calculate the loss for every epoch, which might slow down the model fitting a bit especially in the sequential version. For this reason, we don't need to check the loss values every time. By default we compute it every 10 epochs. Here's a plot of the loss value's progress for our default command-line setup:



Great, now we know how to adjust the optimization parameters to more effectively fit the model. Then How 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 rating scale.

You may have noticed the program printing out three measurements of the RMSE at the end of the command line. 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 first line of defense 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 nuances in the dataset. 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.

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). By convention, we utilize a larger fraction (~50-80%) of the dataset for training and the remaining sections for testing. In this way, the model learns as much latent features from the dataset as possible, without compromising to the pitfall of *overfitting* - when model fits a certain data well enough without good generalizablilty. You may adjust the train-test split ratio 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 data set, the RMSE may fluctuate wildly between runs using the same parameters, simply because of the random selection of observations to be used in the test data.

§ 5. Parallelization & Optimization

We can finally discuss how to speed up the model fitting with parallelization after introducing essential background of the PMF. The functionality for parallelization is indeed the brightest feature of our program, which makes our model training effectively scalable.

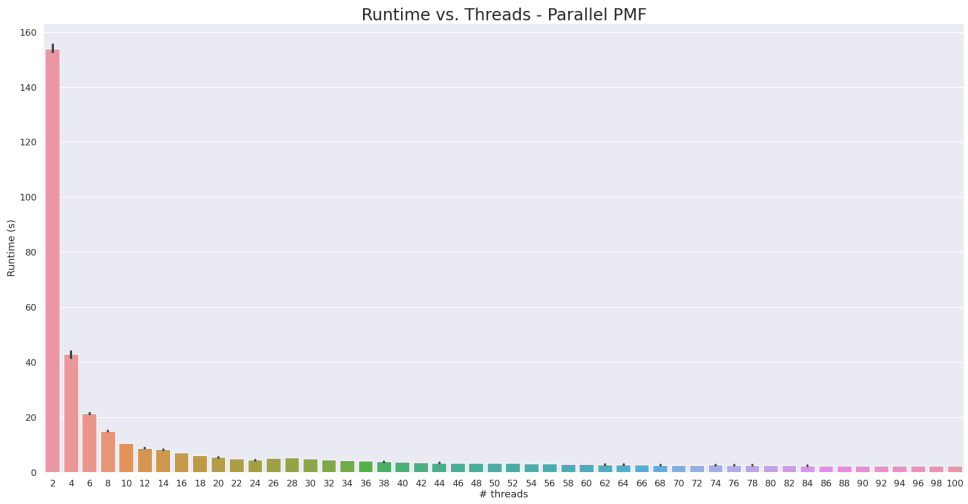
As mentioned earlier, by default, the model fitting will run with multiple threads parallelizing the updates to the vectors. So far we have been running in sequential mode with the `--run_sequential`. We can remove this argument from now. By running the default `./main.tsk -d` command, we use 20 threads for parallelization: the system allocates 19 threads for updating latent user & item vectors and the remaining 1 thread for loss calculation. In other words, we split the training data into 19 batches of equal-size, mutually disjoint submatrices, compute and update parameters in parallel.

The number of threads for parallelization can be adjusted with `--thread` option. For example:

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

It sets 89 threads for model parameter updating and 1 for loss calculation. Due to the nature of the thread allocation in parallel mode --- *at least* one for loss calculation and one for fitting, the minimum number of threads can be specified is 2.

Here's a measure of running time vs. number of threads for our provided dataset on a 12-core regular linux machine. The optimal running time converges after # threads increases to around 50, which takes only around 2 seconds for training. By comparison, the sequential mode for the same dataset takes around 150s and the python implementation takes over 800s! The optimal value for `--thread` varies in different machine and definitely worth tuning.



§ 6. Using the model for recommendations

After learning the latent user preference and item attribute vectors from the dataset, we're now ready to make recommendations. Our program provides a simple interactive way for two types of recommendations: user → item & item → item. Let's see the example of a "recommendation" mode run and we'll break down the details later.

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

Since the terminal in nature has "state-less" feature without storing information between separate runs, we need to remind our program again the items in the *rating* dataset and their *supplementary* information.

Here we use the default input directories for dataset loading. As you might have seen from the terminal, the program asks `Please specify user id:`. Simplify fill in a user id and type return, the program will display the top 10 recommended items that the given user might like the most. Here's a snapshot result in our example run:

```
Loading previously learnt parameters into model...
Please specify user id:
1

Top 10 recommended items for user 1 :

Item: Tokyo-Ga (1985)   Attribute: Documentary
Item: Odd Life of Timothy Green  (2012) Attribute: Comedy|Drama|Fantasy
Item: Distinguished Gentleman  (1992)   Attribute: Comedy
Item: How to Succeed in Business Without Really Trying (1967)   Attribute:
Comedy|Musical
Item: Shaft (1971)      Attribute: Action|Crime|Drama|Thriller
Item: Pursuit of Happyness  (2006)      Attribute: Drama
Item: Searching for Sugar Man (2012)    Attribute: Documentary
Item: Them! (1954)      Attribute: Horror|Sci-Fi|Thriller
Item: Captain Phillips (2013)   Attribute: Adventure|Drama|Thriller|IMAX
Item: From Hell (2001)  Attribute: Crime|Horror|Mystery|Thriller
```

This is the first type of recommendation we provide: recommending items that a user may like from the learned θ vector associated with the given user. We also provide another recommendation plan: recommending items from items. Let's try the example below

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

Similarly, type in an item name and the program will return the top 10 most "similar" items inferred from the model:

```
Please specify item name:
Captain Phillips (2013)

Top 10 similar items to Captain Phillips (2013) :

Item: Escape from Planet Earth (2013)   Attributes:
Adventure|Animation|Comedy|Sci-Fi
Item: From Hell (2001)  Attributes: Crime|Horror|Mystery|Thriller
Item: Brave Little Toaster  (1987)      Attributes: Animation|Children
Item: How to Succeed in Business Without Really Trying (1967)   Attributes:
Comedy|Musical
Item: Pursuit of Happyness  (2006)      Attributes: Drama
Item: Shaft (1971)      Attributes: Action|Crime|Drama|Thriller
Item: Sword in the Stone  (1963)        Attributes:
Animation|Children|Fantasy|Musical
Item: Them! (1954)      Attributes: Horror|Sci-Fi|Thriller
Item: Distinguished Gentleman  (1992)   Attributes: Comedy
Item: Monsters University (2013)        Attributes: Adventure|Animation|Comedy
```

Please keep in mind that the PMF model won't simply recommend items sharing similar attributes (e.g. recommending comedies from comedies), but the recommendation is instead based on interaction effects: A certain cluster of users may love the input item, and happen to like some other movies in common, so the model assumes these common items are "liked" from the common users and consequently must share some latent features in common.

Appendix

Besides the light-weighed command-line interface, we also provide a Python wrapper called "pmPMF", binding with our C++ core Parallel PMF implementation. Users may find it easier to perform analysis, including recommendation and visualization in an interactive Jupyter notebook environment. We're still in progress to fully refine it (to be released in v1.2.0), but here's a current example on how to run and recommend from it. The syntax is almost the same with the command-line program, and both of them uses the executable from our C++ program.

References

- Mnih, A., & Salakhutdinov, R. R. (2007). Probabilistic matrix factorization. *Advances in neural information processing systems*, *20*, 1257-1264
- Niu, F., Recht, B., Ré, C., & Wright, S. J. (2011). Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. arXiv preprint arXiv:1106.5730
- GroupLens Research (2021). MovieLens dataset. https://grouplens.org/datasets/movielens