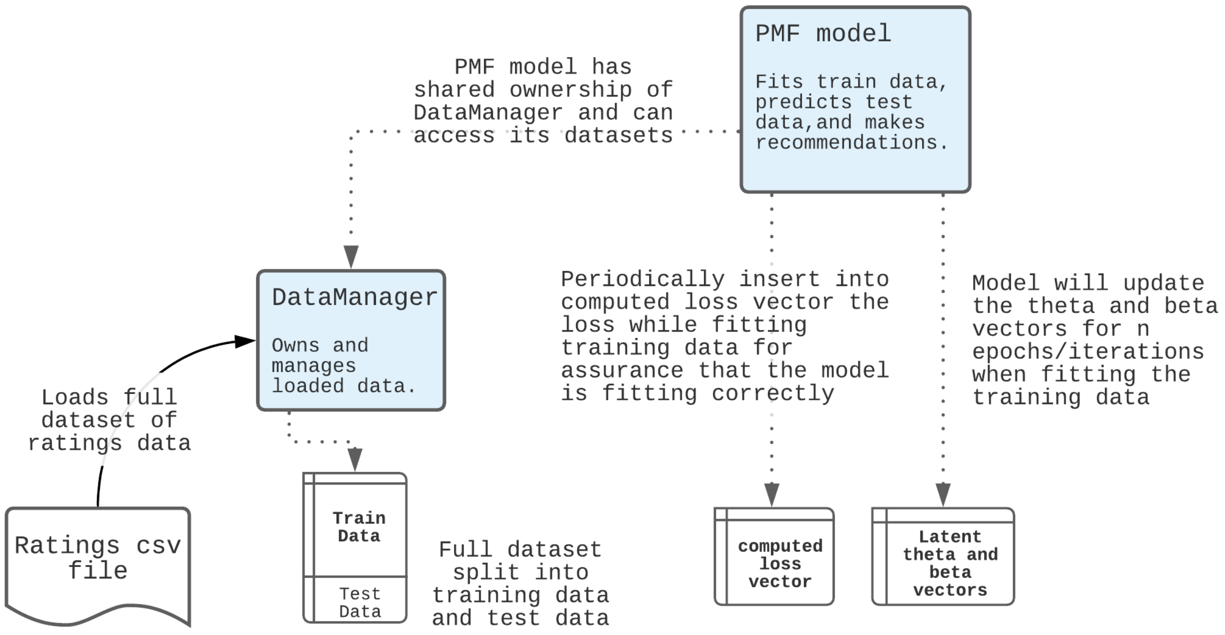
Parallel Probabilistic Matrix Factorization

Anders Geil, Sol Park, Yinuo Jin

I. Design overview

I. High-level overview


PMF model
Fits train data, predicts test data, and makes recommendations.
PMF model has shared ownership of DataManager and can access its datasets
DataManager
Owns and manages loaded data.
Periodically insert into computed loss vector the loss while fitting training data for assurance that the model is fitting correctly
Model will update the theta and beta vectors for n epochs/iterations when fitting the training data
Loads full dataset of ratings data
Ratings csv file
Train Data
Test Data
Full dataset split into training data and test data
computed loss vector
Latent theta and beta vectors

High-level system architecture sketch.

The system consists of two main components: the `PMF model` and the `DataManager`. The `DataManager` handles the loading and preprocessing of the data, supporting various data processing methods used by many types of machine learning applications. The design decisions concerning the `DataManager` component are further discussed in section ***II.v***.

The `PMF model` provides an interface to fit, predict, and recommend over the implementation of the probabilistic matrix factorization. It accesses the data through shared ownership of the DataManager. More specifically, the model accesses the training data in order to perform its fitting, the test data in order to predict, and the entire dataset to make recommendations. The ultimate goal of the model is to update the latent theta and beta vectors during its fitting process, which can then be used for prediction and recommendation.

Fitting the training data is the heaviest computation performed by the model. The model offers both a sequential fitting and a parallelized fitting of the data.

II. Design Considerations

I. Choosing a parallelization scheme

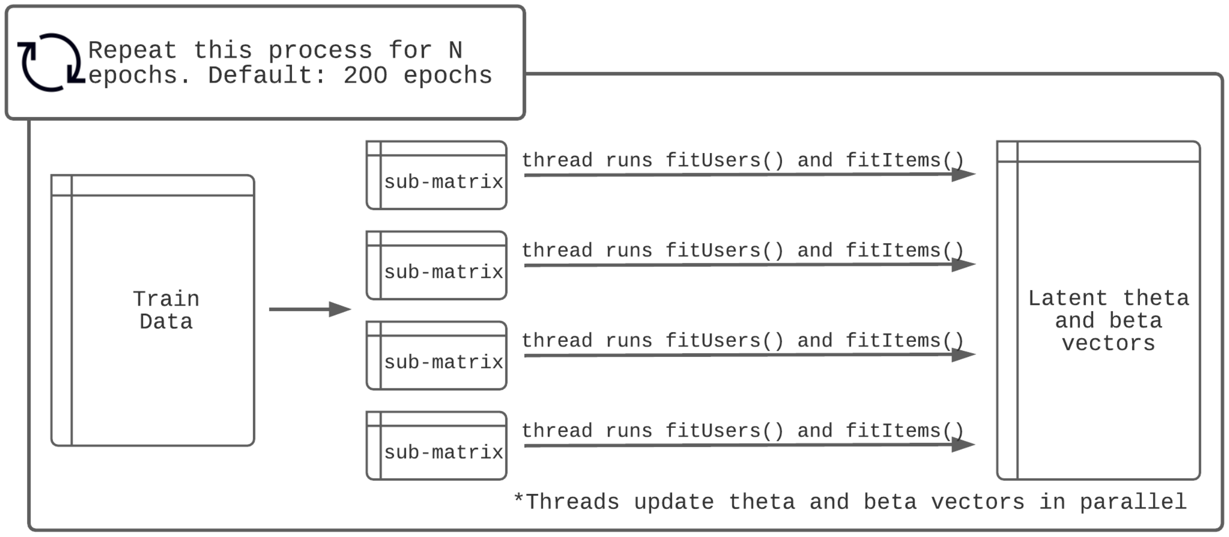
While stochastic gradient descent provides a way to divide the workload of computing gradient updates by approximating the gradients on independent subsets (batches) of the data, we still needed a way to apply these updates to the most current latent vectors efficiently. The key issue is that two threads may simultaneously compute a gradient update on the same latent vector, say θ_i, if both of their batches contain ratings by user *i*. More precisely, the problem is not in computing the gradient updates themselves, but becomes a problem if the two threads try to apply these updates to the most current θ_i at the same time. This would cause a collision, and is a concern unique to the parallel (as opposed to the sequential) model.

To remedy this, the classic approach is to apply locks. Whenever a thread wants to apply an update, it must first acquire a lock such that only one thread can access θ_i at a given time. Due to the large number of small, incremental updates, however, this approach of sequential lock acquisition between threads itself becomes a bottleneck on the performance of model fitting. One approach to lock-free parameter updates, then, is to split the dataset strategically into batches such that only one thread updates the parameters in θ_i for user *i*. That is, all ratings by user *i* are assigned to the same batch and hence the same thread. But partitioning the dataset strategically into batches in such a way leads to two new and separate issues:

- First of all, we would now need to tediously split the dataset into batches *twice*. Once to pick out the rows relating to user *i* in order to update the preference vector θ_i, but then we would also need to partition the dataset a second time to select the rows relating to item *j* to update the attribute vector β_j because multiple users may have rated the same item. While tedious, this is still doable and only incurs an upfront cost once before starting the training procedure.
- Second of all, and more importantly, however, we cannot assume that the dataset is evenly distributed among users and items. In fact, we know from experience that some users are much more active than others, and some items are much more popular (in the sense of number of ratings) than others. Hence, one thread may end up with a disproportionate amount of ratings pertaining to the same user or item, and may take much longer to compute gradient updates for, leaving the other threads waiting for the next epoch to start.

The only way to accommodate this issue, however, is to partition the dataset into evenly sized batches. This leads us back to the initial issue: Does there exist a lock-free approach to applying gradient updates such that collisions do not bias the end result? Luckily, Niu *et al.* (2011) provide exactly such a parallelization scheme, colloquially called the “Hogwild!” algorithm. By exploiting the sparsity in data, they show that not only are collisions exceedingly unlikely to occur in a natural setting, collisions may even function as a regularizer on the model and “accidentally” improve the accuracy when testing on out-of-sample data. By using “Hogwild!”, we permit the training procedure to divide the dataset into evenly sized batches and apply gradient updates with no locking to maximize performance at scale. The main difference between “Hogwild!” and our implementation is that we perform *two* gradient ascent procedures in tandem. For this reason, our results are slightly more experimental in theory. In practice, however, we found that this is not a problem since the computation of gradient updates for the user and item vectors are symmetric and require the exact same number of basic operations, hence taking the same amount of time to compute (on average).

Implementation


Repeat this process for N epochs. Default: 200 epochs
Train Data
sub-matrix
thread runs fitUsers() and fitItems()
sub-matrix
thread runs fitUsers() and fitItems()
sub-matrix
thread runs fitUsers() and fitItems()
sub-matrix
thread runs fitUsers() and fitItems()
Latent theta and beta vectors
*Threads update theta and beta vectors in parallel

Design sketch of fitting submatrices and updating the latent vectors in parallel.

In order to parallelize the batches of submatrices' gradient updates, we chose to utilize the C++ standard `thread` library. This library provided us the means to multithread our fit processing without installing additional dependencies.

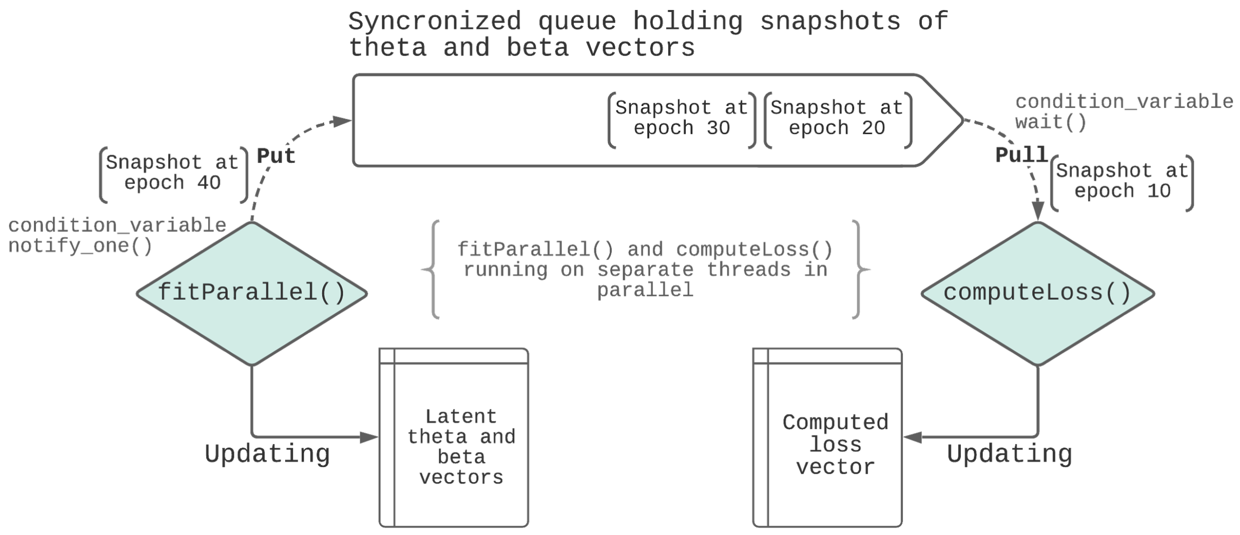
At the beginning of every epoch, the training data is split into N batches, where N is the number of threads provided by the user. Using Eigen, the data is able to be batched "by reference", without additional copying of the data. Then, for each batch of data, a thread is created to fit users and fit items for the given batch. These batches are processed in parallel, updating the latent vectors. At the end of each epoch, these threads are joined, destroying the epoch's pool of threads. This allows for each epoch to remain differentiated from one another, as the next epoch's processing must begin only after the current epoch's processing is completed.

II. Progress verification during model fitting

The next design decision we faced was in choosing between speed and usability. On the one hand, we wanted our model to fit to the data as fast as possible, while on the other hand we also had to acknowledge that end-users may still end up waiting for a significant amount of time, depending on the size of their dataset. From an end-user perspective, then, we may want some kind of intermittent "progress report" to verify that the model is still at work fitting to the data, and is making steady progress. The classic approach to providing this is by periodically printing out a "loss" measure of how well the current state of the model explains the training data, typically at the end of every 10 epochs. In any probabilistic graphical model, such as our matrix factorization model, this can be achieved straightforwardly by computing the joint probability of the data under the most recent iteration of the latent vectors, i.e. evaluating $p(x, \theta, \beta)$. We perform this computation in log-space to circumvent numerical issues as we are typically working with very small probabilities.

The dilemma here arises in that the latent vectors, θ and β, would need to be stable to provide an accurate representation of the current state of the model. In other words, if the worker threads continue to update the latent vectors as we are evaluating the performance of the model on our training data, we may not get a clear picture as to how the current state of the model compares to previous evaluations. We decided to resolve this problem by maintaining a stable copy of the most recent latent vectors upon the termination of every 10th epoch. While copying these vectors incurs a small performance cost by letting our worker threads wait for the copying to complete, the cost to copy the vectors turned out to be noticeably faster than the rest of the loss computation. We deemed the extra cost in space and speed taken to copy to be small enough to provide a reasonable tradeoff, since the rest of the loss computation can be done on a separate thread, unblocking the vector updates. In favor of usability, we would especially expect more advanced users to experiment with the various hyperparameters of our model in order to tune its performance, and this requires a clear window into the behavior of the model during fitting to assess learning and parameter convergence. For basic usage, users with more faith in the default parameter choices may simply increase the number of epochs between each model evaluation to further maximize training speed.

Implementation


Syncronized queue holding snapshots of theta and beta vectors
Snapshot at epoch 30
Snapshot at epoch 20
condition_variable wait()
Pull
Snapshot at epoch 10
Snapshot at epoch 40
Put
condition_variable notify_one()
fitParallel()
fitParallel() and computeLoss() running on separate threads in parallel
computeLoss()
Updating
Latent theta and beta vectors
Computed loss vector
Updating

Design sketch of computing the loss in parallel.

In order to be able to compute the loss while still fitting the model in parallel, we opted to use a synchronized queue, which holds snapshots of the latent vectors at every N (by default 10) epochs. How frequently the user wishes to measure loss can be tuned by the user's provided parameters.
On the main thread, at every 10 epochs of the fitting, the model will take a snapshot of the current state of the latent vectors by making a copy. This copying process must be synchronous as we cannot allow any updates to the vectors when we want a stable snapshot. This snapshot is queued to the standard library's `std::queue`. Then, a `std::condition_variable` is also

used to wake up the loss computation thread to process the items on the queue. Afterwards, this main thread can continue its fitting process in parallel.
On the separate loss computation thread, the loss is being computed by pulling the snapshots off the queue and processing them. This thread uses a `std::condition_variable` to wait for a signal to start popping items off the queue.
As mentioned above, the cost incurred of making a copy of the snapshot showed to be relatively small in comparison to the processing of the loss computation of the snapshot, as the bottleneck was identified to be the loss computation itself. It is also worth noting that just as the copying time will grow with a larger training dataset, the loss computation process will also grow, as the loss computation iterates over the training dataset. This indicates that the copying costs would stay relatively small to the loss computation, even as the dataset grows. This queuing mechanism was able to reduce this bottleneck. Furthermore, using this queue allows the loss computation to use a single thread for the entire fitting process, rather than a new thread per 10 epochs.

III. Model verification after model fitting

As important as model evaluation is to advanced users during model fitting, so is model verification to the general user after the model has finished training. To accomplish this, we decided to provide a set of more generally applicable utility functions useful for evaluating many different kinds of models. The key metric used in our project is the measure of root mean squared error (RMSE), which indicates how many points (on the original rating scale) the fitted model is off by the actual rating scores in expectation. To verify that the model has learned a useful representation of the user preference and item attribute vectors, then, it is useful to compare the RMSE of the predicted values to the RMSE of a naïve model only predicting either the middle of the rating scale (0, since we automatically zero-center the ratings) or the average score in the training set (in case users are more inclined to provide ratings for movies they like than the ones they don't like, or vice versa). A user can then verify the performance of their fitted model by assuring that the error of the fitted model is less than that of the two benchmark measurements. On the small MovieLens dataset, an end-user can expect to see their model achieve a RMSE in the range of 0.9x with benchmark measures above 1.0. A well-tuned model may reach RMSE scores down to approximately 0.85. Finally, as another form of external validation, we also provide our reference (sequential) Python implementation as an additional source of comparison for model accuracy, though we do not recommend using this implementation for larger datasets due to its relatively slow performance.

IV. Choosing a linear algebra library

A significant design decision was made early in the development cycle when we chose to include Eigen3 as an external dependency. We quickly acknowledged that a linear algebra library was required to efficiently handle the many matrix and vector data structures and related operations that our model is intimately reliant on. The specific choice to use Eigen3 was partially made due to its seamless interface with STL containers, and partially out of time constraints leaving little room for consideration of alternative libraries. That said, one of the more useful features in Eigen3 is the ability to reference specific subsets of a (shared) data matrix which we exploited heavily to partition the full training data matrix into batches without having to copy any data into separate data structures. One disadvantage of Eigen3, however, is in its relatively primitive support for boolean indexing and subsetting of matrices and vectors. Unfortunately, this only became apparent to us late in the development cycle when we needed a way to subset data batches by specific user and item IDs. To accommodate this, we instead developed our own helper functions which are not quite as efficient as a native vectorized solution. We later learned that the upcoming version 3.4 of Eigen provides improved support for indexing and subsetting, but at this time we were not willing to upgrade to the experimental version and risk breaking some of our existing functionality so close to the project deadline. We instead decided to postpone the upgrade to Eigen3.4 until its official release, and are eager to see if it provides further performance increases for our model in a future version 1.2.

V. Creating a separate DataManager class

In line with our design priority to create a fast but basic application that is also extendible, we decided to factor out all data management to a dedicated DataManager class. To allow future extensions of the application to include multiple different model classes, we designed an initial DataManager class to ingest a .csv file and convert it to an Eigen matrix in the desired format. We also aimed to make the DataManager class extendible to further preprocessing steps which is a common step performed in many types of machine learning applications prior to actual model fitting. Future maintainers may easily extend the preprocessing procedure by adding further steps beyond our current need for zero-centering of the rating scale and randomly shuffling of the rows in the data matrix. Besides preprocessing, the DataManager class also handles import and export of learned feature vectors. This serves the purpose of allowing end-users the flexibility to continue fitting of their model at a later point after the initial termination of their program. It also permits the separation of model fitting and model prediction as independent applications which we will touch on next.

VI. Supporting model fitting and prediction independently

A common trait for machine learning models is the asymmetric requirements in computation for model fitting compared to its consequent application in prediction. While fitting a model can be a tedious, iterative process of adjusting hyperparameters, assessing the loss curves, and verifying the learned feature vectors, prediction on new data is typically straightforward and computationally lightweight in comparison. By leveraging the DataManager class, we also decided to support model fitting and model prediction as independent applications of our program. Specifically, we designed the DataManager and Model classes such that model fitting is not a prerequisite to prediction by also letting the DataManager support import and export of feature vectors previously learned by the model. In this way, the end-user may skip straight to the prediction phase by utilizing their preferred set of latent vectors, which is a prerequisite to model deployment in most industrial settings. For prediction, we support several options: predicting ratings for a given combination of user and item IDs (the predict() method); recommending the top N new items most likely to be rated highly by a specific user (the recommend() methods); recommending the top N items most similar to a specific item (the getSimilarItems() method).

References

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.