Assignment 3

DSCI 641 - Recommender Systems - Winter 2024

OVERVIEW

In this assignment, you will build and evaluate a hybrid recommender system for books, using data from the UCSD Book Graph collected from GoodReads.

Your recommender will integrate user interaction signals (users adding books to shelves and/or rating books) with book author, genre, and subject data to recommend books. Between this description, the example code, and lecture, I will provide instructions to build your recommender with PyTorch; if you are more comfortable with TensorFlow, you can use it to build an equivalent model, but I will be less able to provide support.

In class, we will introduce PyTorch for training models, and talk about models that are a subset if what you are building for this assignment, applied to MovieLens.

The deliverables for this assignment are:

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

This assignment is due **Saturday, Mar. 3**.

Revision Log

Feb. 27	Update due date in assignment description. Drop item-item baseline.

BACKGROUND

This project integrates ideas from the following three algorithms:

- SVDFeature, a matrix factorization technique that breaks down item (and, optionally, user) embeddings into constituent embeddings from their features, sharing information across items with common features.
- Bayesian Personalized Ranking, the pairwise loss model for learning-to-rank.
- Logistic Matrix Factorization, a pointwise model suitable for implicit feedback.

I am providing the papers for you to read more about these methods if you wish, but we will not be implementing any one of them precisely.

DATA AND SOFTWARE

Conda Environment

This assignment uses LensKit and PyTorch. I have provided two environment files (in the class GitLab) to help with this:

- `environment.yml` installs PyTorch from Pip without GPU support, and takes substantially less disk space. I recommend this one if you need to save disk space, have a slow Internet connection, or don't have an Nvidia GPU.
- `gpu-env.yml` installs the standard PyTorch build, configured for GPU acceleration on platforms that support it. Installing from this environment will take quite a bit of disk space, quite likely 20-30 GB for the software and packages. Some of this can be reclaimed after installation by running `conda clean -a`.

I have also supplied a `requirements.txt` file for use with Pip (including Colab).

Note on GPUs: It is possible, although slow, to run this code without a GPU. You can use Google Colab for GPU access. Recent Mac users can try using MPS, with the most recent version of PyTorch (2.2). If you want to use a TPU on Colab (instead of GPU), you can try using PyTorch/XLA; I have no direct experience with this. PyTorch also supports AMD GPUs (via ROCm); I also have no direct experience with this, but you should be able to get the code working.

Data

I am providing you with a subset of the UCSD Book Graph, a data set collected from public user activity on GoodReads. This data has been reprocessed and integrated with the PIReT Book Data Tools. **Do not redistribute this data** – anyone outside of class needs to obtain it from UCSD themselves.

GoodReads has two levels of items. A *book* is an individual edition of a book. Each printing, edition, translation, format, etc. of a book has a separate book entity. A *work* groups together related books that are versions of the same creative work. Homer's *Odyssey* is a work; each edition of each translation is a book associated with that work. We will be recommending at the **work** level: the user-item interaction files I am giving you are works, and your recommender should recommend works.

The data set includes the following files:

- `gr-item-titles.parquet`: The work IDs and titles, useful if you want to see the titles of the books it is recommending.
- `gr-item-authors.parquet`: The authors for each work. A work may have multiple authors, in which case it has multiple rows in this file.

- `gr-item-genres.parquet`: The genres for each work. Each row consists of a work ID, a genre, and a score. Scores are not comparable between works (more popular works have have higher scores overall). Work-genre pairs with a score of 0 are not included, and we take that to mean the book is *not* in that genre.
- `gr-item-subjects.parquet`: The Library of Congress subject headings for each work. Each row consists of a work ID and a subject heading. Most books have multiple subjects.
- `a3-train-actions.parquet`: User-item interactions for your training data. Each row consists of a user ID, work ID, the time the user first added the work to one of their shelves, the number of times they have added it to a shelf, and the rating (if they have read it).
- `a3-dev-actions.parquet`: Test data for you to use while developing your recommender. This is useful for seeing how you are doing, tuning hyperparameters, etc.
- For A4, you will get another file, `a3-eval-actions.parquet` containing the final test data for you to test your recommender.

The training data consists of user interactions from June 1, 2015 through May 31, 2016, filtered to be a 5-core. The dev and test files are disjoint sets, consisting of user interactions in June 2016 from users who appear in the training data.[1] The supplied notebook is only to provide descriptive statistics as a part of documentation; it cannot be run with the files I have provided (in fact, it is how I created some of them).

THE MODEL

Recall that matrix factorization implements the following scoring function:

$$s(u|i) = b_{ui} + \mathbf{p}_u \cdot \mathbf{q}_i$$

In this design, b_{ui} is the user and item bias terms, $\mathbf{p}_u$ is a k-dimensional vector of user latent features (the *user embedding*), and $\mathbf{q}_i$ is a k-dimensional vector of item latent features (the *item embedding*). These matrices, possibly along with the biases, are the *parameters*, and are learned to optimize the model's ability to predict user activity, as specified by the loss function.

This is, of course, not the only model that can be used. One of the ideas behind the SVDFeature model, along with other related models, is that it is useful to decompose the item's feature vector into sub-vectors derived from features in the item's content or metadata. The result is a hybrid collaborative/content-based recommender that uses both user-item interactions and item content features to learn a recommendation model. If we

[1] Precise details may change. See the notebook for the final description.

do this with books and their authors (represented by the set A_i), we can get the following model:

$$s(i|u) = b_i + b_u + \mathbf{p}_u \cdot \left(\mathbf{q}_i + \frac{1}{|\mathrm{A_i}|} \sum_{a \in A_i} \mathbf{q}_a^{(A)} \right)$$

This model breaks down as follows:

- b_i is the item bias
- b_u is the user bias
- $\mathbf{p}_u$ is the user's latent feature vector
- $\mathbf{q}_a^{(A)}$ is a latent feature vector shared by all books written by a
- $\mathbf{q}_i$ is the item's latent feature vector

When we fit this model, what happens is that some feature relevance is shared among all movies with a particular actor. The item embedding then captures remaining signals that model that *particular* item's potential relevance to users, independent of its actors. This makes it easier to recommend new movies, because we may not have user interactions, but we do have their actors; it also allows us to pool information between books with the same author, using the interactions with popular books by that author to build a better model of less-popular books they wrote. We are taking the average of the book's author vectors, so that books with more authors don't automatically have higher values for the embedding dimensions.

There are quite a few ways we could make this model more sophisticated, but our goal here isn't to make the world's fanciest model – we want to make a useful hybrid recommender. The final, full model you are going to train for this assignment uses authors, genres, and subject headings as the item features for pooling embeddings:

$$s(i|u) = b_i + b_u + \mathbf{p}_u \cdot \left(\mathbf{q}_i + \frac{1}{|\mathrm{A_i}|} \sum_{a \in A_i} \mathbf{q}_a^{(A)} + \frac{1}{|G_i|} \sum_{g \in G_i} \mathbf{q}_g^{(G)} + \frac{1}{|H_i|} \sum_{h \in H_i} \mathbf{q}_h^{(H)} \right)$$

If you want to, you can also experiment with using the book-genre *scores* as part of the model, giving more weight to genres with a higher score for the book, but that is not necessary to complete the assignment.

OBJECTIVE FUNCTIONS

As we have discussed in class, the same core scoring model can be used for different purposes and trained with different loss functions. We can use this model in at least three different ways. We can use it to predicting ratings, by minimizing:

$$\mathcal{L}_{\text{MSE}}(s) = \frac{1}{|R^{\text{test}}|} \sum_{r_{ui} \in R^{\text{test}}} \big(r_{ui} - s(i|u)\big)^2$$

We can use it to estimate the pointwise probability that a user will add a book to their shelf (from logistic matrix factorization) by minimizing the negative log likelihood ($r_{ui} \in \{0,1\}$, α is a positive-example weight to be tuned as a hyperparameter):

$$\mathcal{L}_{\text{NLL}}(s) = -\sum_{u,i} \Big(\alpha r_{ui} s(i|u) - (1 + \alpha r_{ui}) \log\Big(1 + e^{s(i|u)}\Big)\Big)$$

And finally, we can use BPR to estimate the pairwise ranking error:

$$\mathcal{L}_{\text{BPR}}(s) = -\sum_{u, i \in I_u, j \notin I_u} \text{logistic}\big(s(i|u) - s(j|u)\big)$$

For both NLL and BPR, it is not feasible to sum over all pairs (or all triples). Instead, we will estimate the loss with a suitable sample (all positive examples from the data plus a sample of negatives). Note that NLL, like MSE, is a *pointwise* loss: it aggregates the loss function over individual (user, item) pairs. BPR is a *pairwise* loss: it aggregates loss over pairs of users and items. For MSE in implicit-feedback mode, we no longer want to only test the squared error for the rated items, but in theory all items; you should likewise sample negative items instead of testing all items. The `TorchSampledMF` example code from Week 5 demonstrates each loss function. For MSE, convert the ratings using a "confidence level", using the default from the implicit MF paper: train your system to predict a 40 for consumed books and a 1 for negative books.

Think about: for BPR loss, a per-user bias b_u is meaningless. Why?

IMPLEMENTATION

The core of this assignment is an implementation of the hybrid matrix factorization model specified above, capable of being trained with MSE over implicit feedback confidence levels, logistic (NLL), or BPR loss. Implement this as a LensKit algorithm using PyTorch. If you only have time for 2 loss functions, focus on MSE & BPR.

BPR Tips

For implementing with BPR loss, you need to sample negative items for each triple, as we saw in our example code. Use the AdamW optimizer from PyTorch, so you get proper support for regularization (weight decay) in your model.

Logistic Tips

The purpose of the α parameter is to control the relative importance of positive (consumed) examples and negative examples. In the full-data ideal, there are many more negative items than positive items, so a large value of α is necessary to prevent the liked items from being completely overwhelmed by the negative items.

In the logistic matrix factorization paper, the author also talks about sampling negative items to improve the training efficiency. If you sample one negative item per positive item, and use $\alpha = 1$, it is equivalent to treating positive and negative examples equally. Further, you can reuse the sampling logic from BPR: treat each sampled triple as two data points, one positive and one negative.

EXPERIMENTAL SETUP

In this assignment, we are targeting a general front-page recommendation task for established users. All of our test users have at least 5 books in the training set. You will evaluate your recommender using **nDCG** computed over **100-item lists** for the test users. You may wish to subset the training data, particularly for debugging.

For A3, submit comparison of your algorithm against the baselines using the provided development set. Try a few different hyperparameter values (especially # of latent features and regularization weight) on the development set.

Baseline Algorithms

We want to see if our system is any good! Generate recommendations with the following LensKit algorithms to compare to your system:

- basic.Popular
- ~~item_knn.ItemItem (with 20 neighbors, and implicit-feedback mode)~~ [no longer required]
- als.ImplicitMF (with 50 features) (in a real experiment you should tune!)

When we turn off item features in our model and train it with BPR loss, it is BPR, so we don't need to compare against LensKit's BPR (unless we want to check cross-implementation consistency).

Hyperparameter Tuning

For good performance, you will need to do some work to find good hyperparameters for both your algorithm and for the ImplicitMF baseline. These include:

- Embedding dimension (# of latent features). Both your algorithm and ImplicitMF have this.
- Regularization strength. Both your algorithm and ImplicitMF have this, but they are not on the same scale.
- Number of epochs. This isn't necessarily a search problem; more epochs should generally make it improve until it starts over-fitting. A plot of accuracy as you train more epochs is useful to include; if you prepare, one, include it in your report notebook. BPR often continues to improve.

You can do this with grid search: try different feature counts with different regularization strengths. Regularization is pretty magnitude-sensitive, so you can try a few values (e.g. 0.1, 0.01, 0.001), along with several different feature counts (reasonably spaced in the range [10, 500] is a good starting point). Use nDCG on the dev set, possibly balanced with model training time, to select good hyperparameters. Try to leave time to try at least 5 different configurations; if you have time, trying more is good. Hyperparameter tuning will be worth no more than 5% of the final grade, so getting results on a default set of hyperparameters is your first priority.

FINAL DELIVERABLE

Your final deliverable for A3 needs to consist of two pieces: the code itself, and your preliminary evaluation report as a Jupyter notebook. Please provide a PDF export of your notebook for ease of grading, and use matplotlib-based plots (matplotlib, Seaborn, or plotnine) so they export correctly (unfortunately, this means no Plotly).

Code

Submit a zip file that contains your algorithm implementation along with all scripts and notebook(s).

Notebook

Your analysis notebook needs to describe the following:

- A description of your implementation. What decisions did you need to make? What tricks did you need to make it efficient? How long does it take to train your model? (describe your hardware so I have context to interpret this)
- The results of your evaluation, with suitable charts and/or tables. Specify your hyperparameter settings for this evaluation.
- A description of your hyperparameter search strategy, including charts showing its results where appropriate (e.g. a chart showing the dev set nDCG for different embedding sizes).
- A reflection on what you have learned through this project and what you have found difficult.

If there are parts you don't get done, use the notebook also to describe what you did and did not get done, and what you had difficulty with.

Looking Forward

A4 will build on A3, conducting an ablation study on your algorithm and testing additional knowledge-aware techniques.

TIPS FOR SUCCESS

I recommend that you start promptly on this assignment; in class we will be discussing the necessary concepts using the MovieLens data. You will be able use that code as a starting point for your own implementation.

Testing on Small Data

Create a small sample of the training data for debugging your code. This will greatly aid in the work to make sure things actually run.

Use Google Collab

Google Collab will make it significantly easier for you to run experiments efficiently, unless you have good GPU access elsewhere.

Collaborating

You need to submit your own results, but I recommend discussing the implementation on Blackboard, including sharing code snippets where you are getting stuck. You can also collaborate on hyperparameter tuning – share results of that process, tricks for good values you've found, etc.

Compute Resources

Tux nodes can run your model without GPU support.