DSGA1004 - BIG DATA

Final project

- Prof Brian McFee (bm106)
- Junge Zhang (jz3502)
- Jack Zhu (wz727)

Handout date: 2020-04-09

Submission deadline: 2020-05-11

Overview

In the final project, you will apply the tools you have learned in this class to build and evaluate a recommender system. While the content of the final project involves recommender systems, it is intended more as an opportunity to integrate multiple techniques to solve a realistic, large-scale applied problem.

For this project, you are encouraged to work in groups of no more than 3 students.

Groups of 1--2 will need to implement one extension (described below) over the baseline project for full credit.

Groups of 3 will need to implement two extensions for full credit.

The data set

In this project, we'll use the Goodreads dataset collected by

> Mengting Wan, Julian McAuley, "Item Recommendation on Monotonic Behavior Chains", RecSys 2018.

On Dumbo's HDFS, you will find the following files in `hdfs:/user/bm106/pub/goodreads`:

- `goodreads_interactions.csv`
- `user_id_map.csv`
- `book_id_map.csv`

The first file contains tuples of user-book interactions. For example, the first five linrd are

```
user_id,book_id,is_read,rating,is_reviewed
0,948,1,5,0
0,947,1,5,1
0,946,1,5,0
0,945,1,5,0
```

The other two files consist of mappings between the user and book numerical identifiers used in the interactions file, and their alphanumeric strings which are used in supplementary data (see below). Overall there are 876K users, 2.4M books, and 223M interactions.

Basic recommender system [80% of grade]

Your recommendation model should use Spark's alternating least squares (ALS) method to learn latent factor representations for users and items. Be sure to thoroughly read through the documentation on the pyspark.ml.recommendation module before getting started.

This model has some hyper-parameters that you should tune to optimize performance on the validation set, notably:

- the *rank* (dimension) of the latent factors, and
- the regularization parameter *lambda*.

Data splitting and subsampling

You will need to construct train, validation, and test splits of the data. It's a good idea to do this first (using a fixed random seed) and save the results, so that your validation scores are comparable across runs.

Data splitting for recommender system interactions (user-item ratings) can be a bit more delicate than the typical randomized partitioning that you might encounter in a standard regression or classification setup, and you will need to think through the process carefully. As a general recipe, we recommend the following:

Step 3
- Select 60% of users (and all of their interactions) to form the *training set*.
- Select 20% of users to form the *validation set*. For each validation user, use half of their interactions for training, and the other half should be held out for validation. (Remember: you can't predict items for a user with no history at all!)
- Remaining users: same process as for validation.

Step 2
As mentioned below, it's a good idea to downsample the data when prototyping your implementation. Downsampling should follow similar logic to partitioning: don't downsample interactions directly. Instead, sample a percentage of users, and take all of their interactions to make a miniature version of the data.

Any items not observed during training (i.e., which have no interactions in the training set, or in the observed portion of the validation and test users), can be omitted unless you're implementing cold-start recommendation as an extension.

Step 1
In general, users with few interactions (say, fewer than 10) may not provide sufficient data for evaluation, especially after partitioning their observations into train/test. You may discard these users from the experiment, but document your exact steps in the report.

Evaluation

Once your model is trained, you will need to evaluate its accuracy on the validation and test data. Scores for validation and test should both be reported in your final writeup. Evaluations should be based on predicted top 500 items for each user. Metrics

The choice of evaluation criteria for hyper-parameter tuning is up to you, as is the range of hyper-parameters you consider, but be sure to document your choices in the final report. As a general rule, you should explore ranges of each hyper-parameter that are sufficiently large to produce observable differences in your evaluation score.

In addition to the RMS error metric, Spark provides some additional evaluation metrics which you can use to evaluate your implementation. Refer to the ranking metrics section of the documentation for more details. If you like, you may also use additional software implementations of recommendation or ranking metric evaluations, but please cite any additional software you use in the project.

Hints

Start small, and get the entire system working start-to-finish before investing time in hyper-parameter tuning! To avoid over-loading the cluster, I recommend starting locally on your own machine and using one of the genre subsets rather than the full dataset.

You may also find it helpful to convert the raw CSV data to parquet format for more efficient access. We recommend doing these steps early on.

You may consider downsampling the data to more rapidly prototype your model. If you do this, be careful that your downsampled data includes enough users from the validation set to test your model.

Using the cluster

Please be considerate of your fellow classmates! The Dumbo cluster is a limited, shared resource. Make sure that your code is properly implemented and works efficiently. If too many people run inefficient code simultaneously, it can slow down the entire cluster for everyone.

Concretely, this means that it will be helpful for you to have a working pipeline that operates on progressively larger sub-samples of the training data. We suggest building sub-samples of 1%, 5%, and 25% of the data, and then running the entire set of experiments end-to-end on each sample before attempting the entire dataset. This will help you make efficient progress and debug your implementation, while still allowing other students to use the cluster effectively. If for any reason you are unable to run on the full dataset, you should report your partial results obtained on the smaller sub-samples. Any sub-sampling should be performed prior to generating train/validation/test splits.

Extensions [20% of grade]

For full credit, implement an extension on top of the baseline collaborative filter model. (Again, if you're working in a group of 3 students, you must implement two extensions for full credit.)

The choice of extension is up to you, but here are some ideas:

Medium
- *Extended interaction models*: The raw interaction data includes ratings (1-5 stars), but additional information is available, including reviews. Can you use this additional information to improve recommendation accuracy? What about the `is_read` flag for each user/book interaction: how can you use this information?

Easy
- *Comparison to single-machine implementations*: compare Spark's parallel ALS model to a single-machine implementation, e.g. lightfm. Your comparison should measure both effeciency (model fitting time as a function of data set size) and resulting accuracy.

Medium
- *Fast search*: use a spatial data structure (e.g., LSH or partition trees) to implement accelerated search at query time. For this, it is best to use an existing library such as annoy or nmslib, and you will need to export the model parameters from Spark to work in your chosen environment. For full credit, you should provide a thorough evaluation of the efficiency gains provided by your spatial data structure over a brute-force search method.

Hard
- *Cold-start*: using the supplementary book data, build a model that can map observable data to the learned latent factor representation for items. To evaluate its accuracy, simulate a cold-start scenario by holding out a subset of items during training (of the recommender model), and compare its performance to a full collaborative filter model.

Second Easier
- *Exploration*: use the learned representation to develop a visualization of the items and users, e.g., using T-SNE or UMAP. The visualization should somehow integrate additional information (features, metadata, or genre tags) to illustrate how items are distributed in the learned space.

You are welcome to propose your own extension ideas, but they must be submitted in writing and approved by the course staff (Brian, Jack, or Junge) by 2020-05-01 at the latest. If you want to propose an extension, please get in contact as soon as possible so that we have sufficient time to consider and approve the idea.

What to turn in

In addition to all of your code, produce a final report (not to exceed 4 pages), describing your implementation, evaluation results, and extensions. Your report should clearly identify the contributions of each member of your group. If any additional software components were required in your project, your choices should be described and well motivated here.

Include a PDF copy of your report in the github repository along with your code submission.

Any additional software components should be documented with installation instructions.

Checklist

It will be helpful to commit your work in progress to the repository. Toward this end, we recommend the following loose timeline:

- [ ] 2020/04/16: working local implementation on a subset of the data
- [ ] 2020/04/23: baseline model implementation
- [ ] 2019/04/30: select extension(s)
- [ ] 2020/05/07: begin write-up
- [ ] 2019/05/11: final project submission (NO EXTENSIONS PAST THIS DATE)