Book Recommendation Engine

- Sumedha Rai sr5387@nyu.edu
- Hitesh Patel hlp276@nyu.edu

Section 1: Overview

A recommender system is one of the most successful and widespread applications of machine learning used heavily by companies like Amazon, Netflix, YouTube. For our project, we built a book recommendation system using Sparks' alternating least squares (ALS) method to learn latent factor representations for users and items and use those to make book recommendations for users.

We use a large dataset from the book review website Goodreads. The 'interactions' dataset has roughly 230 million user-book interactions and the columns encapsulate information on user ID (user_id), book ID (book_id), if a book has been read by a user (is_read), if a book has been reviewed by the user (is_reviewed) and the final rating given to the book by the user (rating). The 'is_read' column serves as a flag for marking the 'read status' of a book. It takes the value 1 if the book has been read and 0 otherwise. The 'is_reviewed' column follows the same format (1 if the book has been reviewed and 0 otherwise). The ratings column takes values in the range of 0-5, with 5 being the highest rating is assigned to a book. For this project, we did not take into account the user-book interactions where the rating assigned was zero. This was done under the assumption that a zero rating is not an actual rating that can be given to a 'read' book. So the assumption is that the user has not read the book.

Furthermore, we have taken a subsample of the 'interactions' dataset for our analysis. The subsampling process has been described in Section 2 of this paper.

The original 'interactions' dataset has 876,145 distinct users while our subsampled dataset consists of 10,482 unique users. Also, the number of distinct books in the original dataset is 2,360,650 when the 'zero' ratings are retained and we get 2,325,541 distinct books when the 'zero' ratings are dropped from the dataset. For our subsampled data, we have 103,158 distinct books. Table 1 provides the distribution of the ratings in the original dataset (without rating=0) and our subsampled dataset.

Ratings	1	2	3	4	5
Interactions Dataset	1.96	5.92	22.29	35.87	33.96
Subsampled Dataset	2.11	6.92	26.71	37.54	26.72

Table 1: Distribution of ratings as a percentage of the total ratings in the dataset. The 'Interactions Dataset' is the subset of the original dataset where rating =0 has not been taken into account.

The recommendation system built as part of this project ensures that we capture and continuously calibrate the reading preferences of the users by learning from the extant data available to us. This enables the website to make better recommendations to its user and increase user interaction on the platform.

Section 2: Data Preprocessing

The reader-book interactions dataset consists of ~228.6 million ratings. We removed all the rows with a rating of zero to create a new dataset. We further sub-sampled this dataset by taking 1% of this new dataset. Our subsampled dataset contains ~172k ratings. The subsampling was done because of resource constraints on the shared clustered. We split this subsampled dataset into train, validation, and test sets such that the training set consists of all the users while the test and validation set each consist of 20% of the users i.e. 60% of the users in the training set have all their interactions and the remaining 40% users have half of their interactions in the test and validation sets. There is no overlapping of users in the validation and test sets. We also ensure that all the books in the validation and the test set are present in the training set. These datasets were stored in parquet format to be used later. Furthermore, we fixed a seed for all the random operations so that the results are reproducible.

Section 3: Implementation

Fitting ALS (Alternating Least Squares) model

ALS is a matrix factorization technique that decomposes a matrix X into 2 matrices, U and V in a parallel fashion. It alternates between optimizing the loss function for both U and V i.e. it optimizes U by treating V as a constant and then optimizes V by assuming U as a constant. For building recommendation engines, one of the algorithms used is matrix factorization. The user-item interaction matrix consists of ratings given by a set of users to a set of items and the goal is to predict ratings given by a user to the items that the user has not interacted with yet. The loss function that ALS optimizes is:

$$J(U,V) = \sum_{\{i,j\}} c_{\{i,j\}}\left(X_{i,j} - U_i^T V_j\right)^2 + \lambda\left(\sum_i \left||U_i|\right|^2 + \sum_i \left||V_i|\right|^2\right)$$

λ is the regularization parameter and U, V are the latent matrices for users and items respectively.

We used pyspark.ml.recommendation library to fit an ALS model to our dataset. The hyperparameters that we experimented with are:

1. Rank: 10, 15, 25, 50, 100
2. $\lambda : 10^{-3}, 10^{-2}, 10^{-1}, 1$

These combinations were tested on the validation set and the models were compared using MAP (mean average precision) estimated on 500 items predicted for each user. The business application behind building a recommendation engine is to recommend items to the users. A good recommendation engine predicts items that the users will like/click on. Thus, to assess the performance of any recommendation system, we need to calculate its precision because that captures how many of the predictions made by our model are good. We calculate the precision for the top 500 predicted books recommended for each user. We choose MAP and not precision@k because MAP also takes into account the ranking among the predicted items whereas precision@k only takes into account the presence/ absence of an item the predicted set. [3]

Using the pyspark.ml.recommendation library, we can directly get 500 tuples of predicted items for each user along with their corresponding predicted ratings. We extract the 500 items from these tuples and put them in a new column using pyspark.sql.functions.explode. Windows

function allows us to operate on a group of rows while still returning a single value for each input row. We use this function by partitioning over the user_id column. We then use the pyspark.sql.functions.collect_list function to combine all the predicted books for a user in a single list and make a column out of it. We iterate the process for the actual books read by the user. Finally, the dataframe with user_ids, predicted books, and actual books column is fed to pyspark.mllib.evaluation.RankingMetrics function as an RDD that calculates the MAP of the model. The results of hyperparameter tuning are summarized in Table 2. Our best model has a MAP score of ~0.00079 on the validation set with rank = 100 and regularization parameter = 0.001.

Rank	Regularization	MAP
10	1	3.966E-05
10	0.1	1.528E-04
10	0.01	3.327E-04
10	0.001	3.869E-04
15	1	5.818E-05
15	0.1	1.522E-04
15	0.01	3.620E-04
15	0.001	4.599E-04
25	1	6.690E-05
25	0.1	1.515E-04
25	0.01	2.577E-04
25	0.001	7.855E-04
50	1	2.573E-05
50	0.1	1.751E-04
50	0.01	2.062E-04
50	0.001	6.063E-04
100	1	4.212E-05
100	0.1	1.880E-04
100	0.01	2.354E-04
100	0.001	**7.995E-04**

Table 2: Results of hyperparameter tuning on the validation dataset with top 3 MAP scores highlighted

On the test set, our best model gives us a MAP score of ~0.002 (rank=100, $\lambda = 0.001$)
In the next section, we have implemented two extensions on top of the baselines collaborative filter model.

Section 4: Extension 1 - Comparison to single machine implementations

We used the LightFM library to fit a recommendation model based on explicit feedback to the same train and test splits created as described above. We had to do create sparse matrices out of our pandas dataframes before feeding them to the model. We use the Weighted Approximate-Rank Pairwise (WARP) loss function because we only have positive interactions in our dataset and this loss function optimizes Precision@k indirectly as an evaluation metric. The LightFM library doesn't have MAP (mean average precision) as a built-in evaluation metric, so we used Precision@k to evaluate our model. We didn't use any separate threads while training the model and we use the hyperparameters of our best model from validation done on cluster implementation. We subsample from the 80% users in the train dataframe to create new training sets of different sizes. We fit the recommendation model from LightFM on these training sets separately and evaluate it on the same test set. We measure the amount of time taken in different steps for these training sets.

The analysis results are shown in Table 3.

	Data Preprocesing	Model Fitting	Calculating Precision@k	Precision@k
Fraction			*(time, in secs)*	*(value)*
0.16	0.08	45.57	817.57	0.0167
0.24	0.07	101.74	2190.81	0.0053
0.36	0.07	53.24	2599.80	0.0014
0.55	0.04	50.15	3000.88	0.0012
1	0.03	52.34	3808.67	0.0010

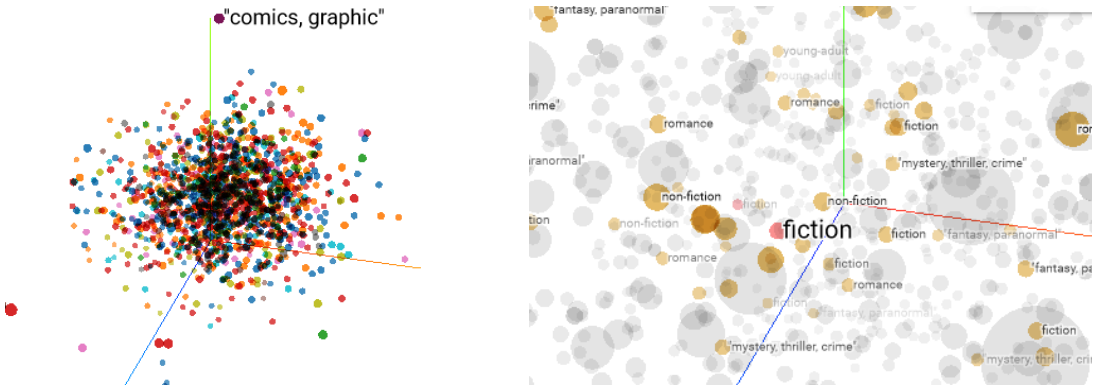
Table 3

On Hadoop, the time taken for each step is reported in Table 4.

Task	Time Taken (in seconds)
Loading The Datasets	33.47
Model Fitting	250.18
Calculating MAP	197.91

Table 4

Section 4: Extension 2 – Exploration



We have used t-Distributed Stochastic Neighbor Embedding (t-SNE) technique that is used for dimensionality reduction to visualize the embeddings learnt as the latent factors to make recommendations. For visualization, we made use of the item latent representations obtained from the best model. We used genre corresponding to each book by mapping the genre to the item latent representation for each book. We have made use of TensorFlow's Embedding Projector by adding the latent factors and 10 different genres for generating the plots. We visualized 3000 data points by tweaking different combinations of tSNE's hyperparameters. We tried perplexity values ranging from 5-60 and epsilon values ranging from 0.01-1 while keeping the number of iterations as constant. The output of tSNE visualization was different for each combination. The visualization for tSNE depends heavily on hyperparameter tuning. Hence, given the plot, it may appear that the model has not learnt well but it may very well be the absence of having isolated the right hyperparameters for tSNE.

Contribution Statement:

Each member has made an earnest attempt to contribute equally to the project according to his or her abilities and personal situations. Sumedha worked on writing the code for data preprocessing and model fitting. Hitesh worked on writing the code for evaluation. Both worked equally on running the scripts for hyperparameter tuning and extension.

Github File Descriptions:

- read_data.py: Reads data from HDFS and creates the final train, validation, and test datasets and saves them in parquet format
- model_als_hyperparam.py: Uses ALS to train data and does hyperparameter tuning on the validation dataset
- model_als_test.py: Uses the best model configuration to calculate MAP scores for the test data
- extension1_lightfm.ipynb: Implements a recommender model using LightFM library for training datasets of different sizes
- extension2_exploration_tSNE.ipynb: Visualization of learnt latent representation of books
- report.pdf: Final report

References:

1. Matrix Factorization Techniques for Recommender Systems https://dl.acm.org/doi/10.1109/MC.2009.263
2. Large-Scale Parallel Collaborative Filtering for the Netflix Prize https://link.springer.com/chapter/10.1007%2F978-3-540-68880-8_32
3. Evaluation Metrics - RDD-based API https://spark.apache.org/docs/latest/mllib-evaluation-metrics.html
4. LightFM documentation: https://making.lyst.com/lightfm/docs/home.html