BookDB Book Recommendation Engine

Introduction and Motivation

As book readers, we are always on the lookout for new books that quench our thirst for an intellectual or entertaining experience. Sadly, it is quite rare to find a great book recommendation. Our goal is to design and implement a system using machine learning to recommend books to users based on their reading history. For this, we will use the public datasets from Goodreads gathered and maintained by researchers at UCSD. We will focus on content-based and collaborative filtering in addition to a reranking component. Finally, we aim to use these models to power a consumer-facing product where users add books to their library and get recommendations based on that.

EDA and Preprocessing

We primarily use 2 Goodreads datasets for our project: book metadata, and user-book interactions. Book metadata consists of intrinsic information about books: title, ISBN number, description, author(s), popular book shelves, and similar_books which is an array of books that Goodreads has determined as similar to the book. User-book interactions consist of an interaction between a user and a book at a timestamp, often accompanied by a rating and/or a review.

There is a third dataset called books_works which maps multiple books to a single work. For example, different languages of a book, different editions, etc. are counted as different books but in the end represent a single work. The books_works dataset provides a best_book_id attribute for each work which we used to de-duplicate the initial books dataset.

The original datasets included 2.3 million books and 228 million interactions. This scale of data already slowed our progression considerably in the EDA stage for trivial tasks like loading and converting the data into different formats, but made any sort of training infeasible. Therefore, we decided to reduce the data by keeping users with 100+ interactions and books with 500+ interactions. In addition to meaningfully reducing the size of the data (~17k books, ~50M interactions), this had a natural effect of cutting the long tail of books that weren't popular and users that weren't very active, potentially improving the results of the training.

Content-Based Filtering

The aim of content-based filtering is to suggest new items for a user that are similar in content and semantics to items the user has already liked. This helps us combat the cold start problem for books that haven't had many interactions. For this, we thought to use pre-trained sentence transformers (SBERT models) that create a dense vector embedding from text that captures semantic meaning. With items being represented as vector

embeddings, we could query similar items to a user's library by computing the mean of the embeddings of their library items and sorting unread books by highest cosine similarity.

Feature Engineering

For fine tuning a SBERT model, we need to construct triplets in the form of (anchor, positive, negative), where anchor is the text representation of the current book, positive is the text representation of a similar book, and negative is the text representation of a non-similar book (picked randomly from the books that are not similar to the anchor). We use the similar_books attribute of the books dataset to construct these triplets.

The text representation of each book is following format: "Title: {title} | Genres: {genres} | Description: {description} | Authors: {authors}" where the genres were extracted from the popular_bookshelves of the books dataset.

Data Splitting

After generating ~55k triplets, we split the triplets into a 80-10-10 train-val-test split. Although we didn't extensively test the optimal split due to time constraints, this follows a rule of thumb and ~5k examples provide meaningful performance evaluation.

Baseline Evaluation

In order to establish a baseline performance, we tested the base SBERT model all-MiniLM-L6-v2 on our test set, using cosine similarity accuracy with a triplet margin of 0.5 as our evaluation metric. We obtained a cosine similarity accuracy of 1.67% which is very bad.

Finetuning

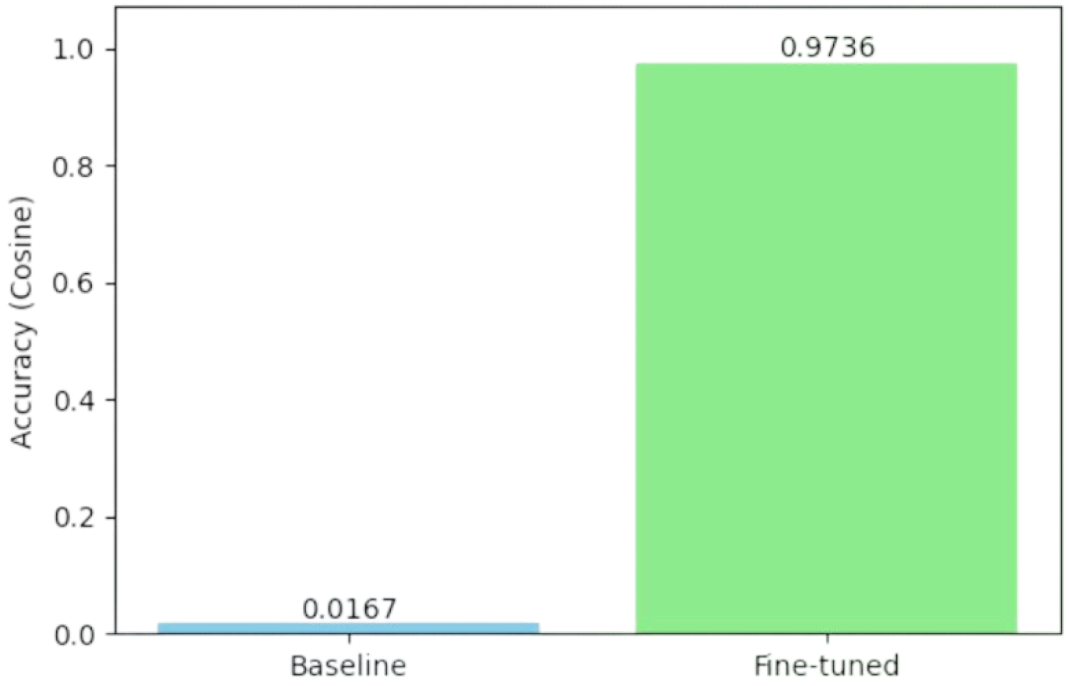
Clearly, the base model didn't perform well on our dataset. To finetune it to our dataset, we used the TripletLoss function as our loss function and cosine similarity as our distance metric. Since we didn't have the resources for hyperparameter tuning, we adhered to the ones most commonly used in the literature:

- Batch_size = 32
- Epochs = 4
- Learning_rate = 2e-5
- Triplet margin = 0.5
- Warmup steps = 10% of total steps

The finetuning took ~2 hours to complete on a Macbook Pro M4 with 48gb RAM

Results

Once the model was ready, we evaluated it on the same test set as the baseline and obtained a cosine similarity accuracy of 97.6%.



Collaborative Filtering

To capture user's interest beyond the explicit content of their interacted books, we introduced a collaborative filtering component where we aim to recommend books that other users with similar interests to a certain user have liked.

We started with graph neural network models for collaborative filtering such as NGCF and LightGCN but moved to the simpler Neural Collaborative Filtering framework introduced in 2017 due to computational constraints. We used the pytorch implementation of the NCF paper with our own modifications to tolerate a larger scale dataset.

Data Splitting

For splitting the data, we took out the latest interaction for each user (sorted by timestamp) as a holdout test set.

Feature Engineering

Since the NCF framework is designed for implicit ratings (interactions as opposed to explicit ratings of e.g. 1-5), we filtered interactions of explicit ratings of 4 and 5 and labeled them all as 1 (interaction exists). We then constructed triplets of (user, item, label) creating 4 negatives (items the user has not interacted with) for each positive (item the user has interacted with). The label indicates the existence of an interaction between the user and the item.

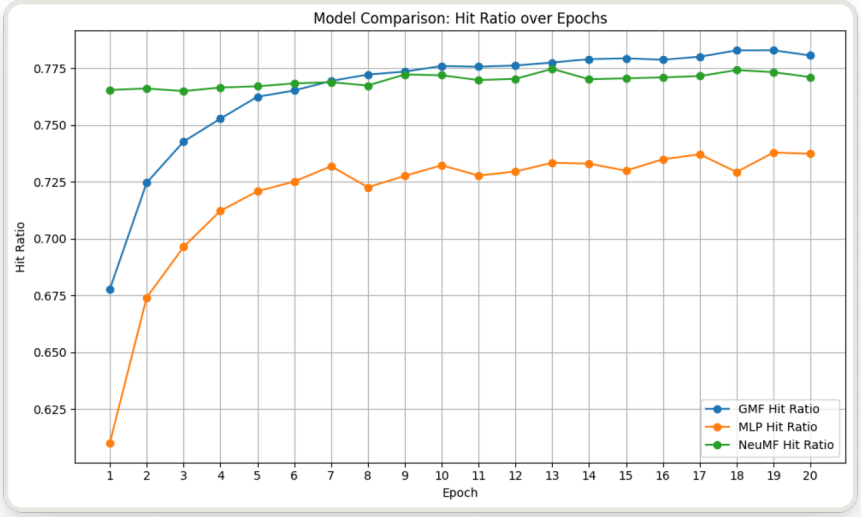
Training

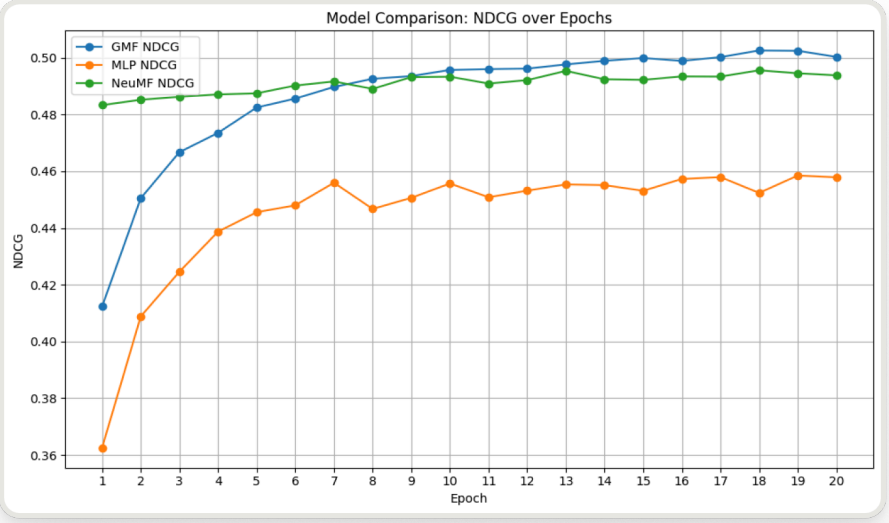
The NCF framework consists of training a Generalized Matrix Factorization model (GMF) , which is a dot-product calculation between users and items followed by a layer of neurons to learn the weights, and a Multi-Layer Perceptron (MLP) model to learn an arbitrary function (as opposed to the dot product) which might improve the results and fusing the learned embeddings together.

Due to resource constraints, we used the hyperparameters supported by the literature for the training:

- Epochs: 20
- Optimizer: Adam
- Weight initialization: Gaussian
- Adam Learning Rate: 1e-3
- Number of negatives: 4
- Latent dimensions: 32
- MLP Layers: 64, 64, 32, 16, 8

Results





The GMF and the fused Neumf models performed similarly with the GMF model slightly outperforming the fused model at NDCG of ~0.5 (predicted item in 3rd rank on average) and a hit rate of ~0.78 (target item is present in top 10 predictions 78% of the time). We suspect this is due to the fact that our dataset can be sufficiently learned through linear relationships (using the dot product for matrix factorization).

Reranker

Modern recommendation systems often utilize a two-stage architecture:

1. Candidate Retrieval: Quickly selecting a broad set using low-latency models (e.g., GMF embeddings).
2. Candidate Reranking: Employing richer features and powerful models to refine recommendations.

This section outlines our reranking method, including deterministic filtering rules and a BERT-based cross-encoder, excluding candidate generation previously discussed. For a more detailed

Custom Logic for Filtering and Diversity

We apply deterministic filters before reranking to improve recommendation quality:

- Removing duplicates and books already read.
- Implementing Maximal Marginal Relevance (MMR) to balance genre/author diversity and relevance.

Initially, recommendations sometimes included duplicates or overly similar content. The implementation of these rules significantly enhanced the variety and quality of recommendations without modifying underlying models.

Reranking Model Selection

Our reranking strategy required a balance of performance and computational efficiency:

Method	Pros	Cons
Rule-based/MMR	Fast, diverse, easy implementation	No learning capability
LightGBM (LambdaMART)	Fast, interpretable	Extensive feature engineering
Cross-Encoder	Deep interactions, minimal feature engineering	Higher inference cost

We selected a cross-encoder model due to its superior capability in capturing nuanced semantic interactions and minimal feature engineering requirements, which aligns well with our dataset constraints.

Why ms-marco-MiniLM-L-6-v2?

This model, distilled from BERT using MS MARCO passage ranking data, offers:

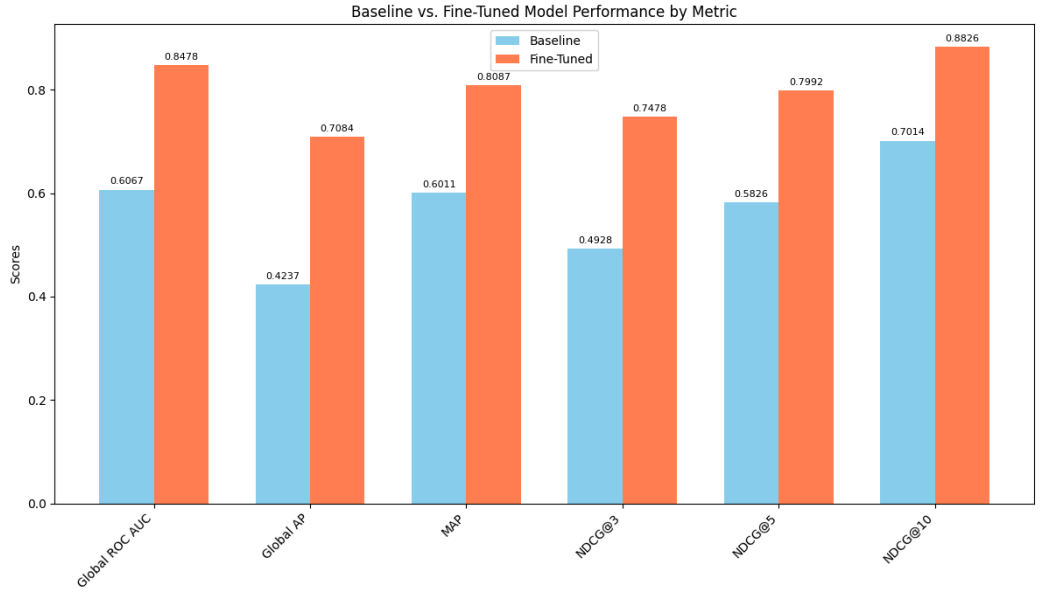
- **Efficiency:** Compact (60MB) compared to full BERT (420MB), providing near-equivalent performance.
- **Fast Fine-Tuning:** Achieves optimal performance within 2-3 epochs.
- **Operational Suitability:** Fits efficiently within our deployment infrastructure.

Feature Engineering

Cross-encoders require paired inputs (user context and book descriptions). Our features included:

- **User Context:** Summarized user preferences (top books and genres). Limited user metadata posed a challenge but proved sufficient for model adaptation.
- **Negatives Sampling:** Due to only explicit positive ratings (≥4), we created negatives via random sampling at a 1:3 ratio to balance dataset size and contrastive signals.
- **Context Leakage Prevention:** Implemented "leave-one-out" strategy for constructing user context, preventing the model from exploiting direct title matches.
- **Data Constraints:** Input length capped at 384 BPE tokens; dataset subsampled (50K positives, 150K negatives) to manage computational constraints and ensure diverse contexts.

Fine-Tuning Results



Fine-tuning significantly enhanced model performance:

- ROC AUC: 0.607 → 0.848
- Global AP: 0.424 → 0.708
- MAP: 0.601 → 0.809
- NDCG@3: 0.493 → 0.748; NDCG@5: 0.583 → 0.799; NDCG@10: 0.701 → 0.883

These results confirm:

- **Feature Relevance:** User and book representations effectively captured relevance signals.
- **Data Size and Quality:** Positive-negative sampling strategy successfully enabled learning nuanced preferences. Despite downsampling the dataset was large enough to significantly improve performance
- **Model Suitability:** Cross-encoder architecture demonstrated strong capability in nuanced semantic matching, confirming the value of fine-tuning on domain-specific data.

Model Inference and Usage

Once the content-based and collaborative filtering models were trained, we used them to generate the embeddings for items and users and stored the embeddings in the Qdrant vector database.

This allows us to obtain recommendations simply by performing similarity search over the mean of the vector embeddings of items or users.

Devops

For the DevOps part of this project, we wanted to keep things simple and easy to use.
To ensure functionality of our models and project we opted to use a simple Docker Container Setup which would containerize each application. This allowed us to perform local dev

testing which would work on any machine and function similar to how actual deployment would be.
To deploy this application, we chose Portainer as a Container Orchestration Manager on a low cost server. This allowed us to perform CI/CD by simply pushing to a ‘prod’ branch. Our application would automatically be pulled by Portainer. Changes made would be recognized, tested and built. Any errors would be logged by a Slack/Discord Bot we had set up. Setup would only be restarting the container which had been changed resulting in minimal downtime. For User Metric logging this would also allow us to spin up extra containers in the case of increased loads.

Challenges and Limitations

As mentioned earlier, the size of the data combined with limited compute resources limited our ability to pursue more advanced training methods like graph methods or perform hyperparameter tuning for our models. Better results could be achieved with more advanced training models and hyperparameter tuning.

Furthermore, the lack of demographic user data limited us to be able to only use user-book interactions for recommendation which becomes challenging when users have interacted with thousands of books of all sorts of different genres and types.

Future Work

Given our challenges with limited sources and types of data, especially user demographics and more nuanced types of interactions, and the fact that we have built a consumer-friendly app, we can expand on this project by logging and collecting user data and behaviors which would later be used for training more advanced and nuanced models for recommendation, given the computational resources. For example, sentiment analysis models could be used to detect a user’s feelings toward a book, author, or genre which could be used in a feature engineering step in the pipeline.

Furthermore, there is an interesting space for implementing and automating MLOps. Particularly, a regular check for distribution shift based on statistical analysis of model performance and automated retraining triggers.

References

NCF Paper: https://arxiv.org/abs/1708.05031
NCF Pytorch implementation: https://github.com/yihong-chen/neural-collaborative-filtering
SBERT documentation: https://huggingface.co/blog/train-sentence-transformers