Project Report
Collaborative Filtering

Data Science Lab

Master IASD
Université PSL

The Shawshank Redemption
Alejandro Jorba, Luka Lafaye, Newman Chen

09/10/2024

Context

For this project, we were tasked with implementing two different approaches to the collaborative filtering problem. Briefly, this is a problem common to many recommender systems, in which one recommends new items to users which they have not yet seen/rated. This is done by using the known ratings of other users in order to fill out a ratings matrix, **R**, which is of dimension $\mathbb{R}^{mxn}$. This large matrix is originally sparse, only filled with the already submitted ratings of users of items they are familiar with. When filled out using a given method, the completed matrix, specifically the values that were previously not filled in, will serve as predictions for how the user would rate that given item and thus enable personalized recommendations by a recommender system.

Methodology

Matrix Factorization

The matrix factorization method involves calculating two lower rank matrices, $\mathbf{U} \in \mathbb{R}^{mxk}$ and $\mathbf{V} \in \mathbb{R}^{nxk}$, that when multiplied together approximate the **R** matrix. k is a model parameter and is known as the number of latent factors. In its most basic implementation, these two matrices are randomly initialized and then updated using gradient descent to minimize the value of the following loss function:

$$J(U,V) = \frac{1}{2}||R - UV^T||_F^2$$

In practice, however, not all of the values of the **R** matrix are known, obviously. Therefore, after initializing **U** and **V**, we then find all the indices (i, j) where the value $\mathbf{R}_{ij}$ of the ratings matrix is observed, which we place in a set S. Gradient descent is implemented with the difference between **R** and our approximation of **R**, $\mathbf{E} = \mathbf{R} - \mathbf{UV}^T$, which is only computed for the entries with existing values in **R**. The matrices **U** and **V** are updated until a certain convergence condition is met, which we simply based off of epochs:

$$u_{iq} \leftarrow u_{iq} + \alpha \cdot \sum_{j:(i,j)\in S} e_{ij} \cdot v_{jq} \text{ and } v_{jq} \leftarrow v_{jq} + \alpha \cdot \sum_{i:(i,j)\in S} e_{ij} \cdot u_{iq}.$$

We first implemented vectorization through the numpy library, which led to faster runtimes. Essentially, we set all of the entries in **E** not in S to zero. In this way, we are able to compute the loss by:

$$J(U,V) = \frac{1}{2}(tr(R^TR) - 2\,tr(R^TUV^T) + tr(VU^TUV^T))$$

and update **U** and **V** as follows:

$$U \leftarrow U + \alpha EV$$
$$V \leftarrow V + \alpha E^TU$$

With the further addition of L2-regularization, aimed to prevent overfitting, the loss function and the matrix updates were:

$$J(U,V) = \frac{1}{2}(tr(R^TR) - 2\,tr(R^TUV^T) + tr(VU^TUV^T) + \lambda\,(tr(U^TU) + tr(V^TV)))$$
$$U \leftarrow U(1 - \alpha \cdot \lambda) + \alpha EV$$
$$V \leftarrow V(1 - \alpha \cdot \lambda) + \alpha E^TU$$

We then added a user and movie bias term. With this approach, the main change is that a user's rating is explained by both the product $\mathbf{UV}^T$ and two other terms. The main idea behind this approach is that classic MF is not able to capture systematic differences such as users who tend to give higher/lower ratings overall or items that are generally rated higher. Although the classic formulation for the inclusion of these bias terms involves calculating as any other parameter, separate from **U** and **V**, we decided to implement an alternative but mathematically equivalent formulation. We increased the dimensions of **U** to m x $(k + 2)$ and of **V** to n x $(k + 2)$, and set up the $(k + 2)^{th}$ column of **U** and the $(k + 1)^{th}$ column of **V** to contain only 1s. The loss function and the update methods remain unchanged.

Though these modifications improved the RMSE scores substantially, we decided to introduce other techniques, especially postprocessing. Since the outputs are half integers in the range [0.5, 4.5], we wanted to fit the values more closely, aiming to improve both accuracy and RMSE. We started with a naïve rounding method, calling the following function on the $\mathbf{R}^{\wedge}$ matrix, element-wise: `np.ceil(matrix * 2) / 2`.

From there, we incorporated a more sophisticated approach that factors in the distributions of the input. As noted in class, the distribution is skewed right, with higher ratings being more common (and 4 being the most common rating). We wanted our output to be more reflective of this, so we implemented a simple method to transform our output, which scales the output by the mean and standard distribution of the input. We avoided more complex methods with the fear of overfitting, since the distribution is already very sparse.

The final addition to matrix factorization momentum, which helps to smooth and speed up the gradient descent process. In this approach, we started by initializing the momentum terms for **U** and **V** with all zeros, **vU** and **vV** respectively. Then, we iteratively update the velocities and the matrices, as follows:

$$vU \leftarrow m \cdot vU + \alpha(EV - \lambda U)$$

$$vV \leftarrow m \cdot vV + \alpha(E^T U - \lambda V)$$
$$U \leftarrow U + vU$$
$$V \leftarrow V + vV$$

Strengths

- MF is straightforward to implement and can work well with dense data.
- By projecting users and items into a shared latent space, MF captures patterns such as user preferences and item properties in a continuous vector space.

Limitations

- MF is a transductive method, meaning it requires a full user-item interaction matrix during training. It performs poorly when dealing with new users or items
- Standard MF only models the interactions (ratings) between users and items. It doesn't directly take into account side information which can be useful for improving predictions.
- MF does not consider the complex relationships between users, items, and their neighborhoods (e.g., users who rated similar movies), which limits its ability to capture rich interactions beyond the user-item matrix.

Graph-Based method

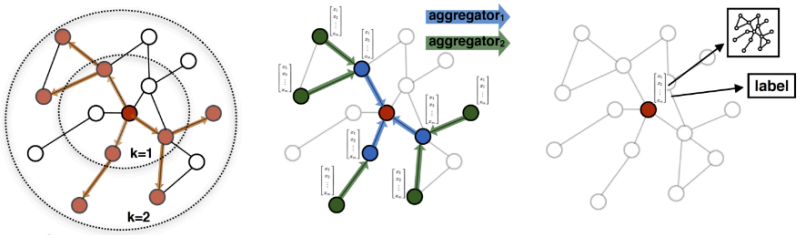
For our method, we create a heterogeneous graph where **nodes** represent users and movies, and **edges** represent interactions between said users and movies. We initialize our nodes with feature vectors, and create an edge between every user and the movies they have rated, where the labels for these edges represent the rating scores. We create the graph data structure using **PyTorch Geometric's HeteroData** class.

The feature vectors we create for our nodes for users are one-hot encodings of size 610, as there are 610 users total in R. The feature vectors for movies are embeddings that include metadata of all movies, including Title, Year, Rated, Genre, imdbRating, imdbVotes, Language, Country, Plot, Runtime. These were obtained after days of scraping and manually verifying/modifying the matrix metadata we import in our notebook.

In order to actually predict the ratings users give to new movies, we need to perform an edge prediction task, where we predict the label of these edges. Given a specific user node and a movie node, the model predicts the edge label between them, if it doesn't already exist.

Our model contains two parts, an encoder and decoder. The encoder first aggregates neighborhood information and computes embeddings for users and movies using **GraphSAGE** layers. During the encoding process, each node's embedding is updated by aggregating features from its neighboring nodes, both users and movies.

The **encoder** structure itself consists of two SAGEConv layers, which both serve to aggregate information from every node that is '1 hop' away. In this case, a user has movies as neighbors whereas a movie has users as neighbors.



1. Sample neighborhood
2. Aggregate feature information from neighbors
3. Predict graph context and label using aggregated information

The general algorithm for the GraphSAGE convolutional operation is:

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

Where h_v^k is the embedding of node v at layer k, $N(v)$ are the neighbors of node v, W_k is the learnable weight matrix at layer k, and σ is a non-linear activation function. In our notebook LeakyReLU worked best (compared to other options like ELU and ReLU).
The encoder outputs final embeddings for both users and movies, incorporating information from their respective neighborhoods. We also applied a dropout layer in the encoder, by randomly setting a fraction of the neuron activations to zero during training. It helped prevent overfitting and generalize better to unseen data (drop of up 0.2 in validation RMSE for equivalent number of epochs).
The **decoder** predicts the rating (edge label) between user and movie nodes. It takes the embeddings of a user and a movie, concatenates them, and passes them through a linear layer to predict the rating:

$$Z = \text{CONCAT}(z_{user}, z_{movie})$$
$$R' = LeakyReLU(W_d \times Z)$$

With

- z_{user} and z_{movie} are the embeddings of the user and movie nodes
- W_d is the weight matrix of the linear layer of the decoder
- R is the predicted rating of the edge that joins the user and movie

The model is trained to minimize the MSE (between R' predicted ratings and true ratings R, the actual edge labels). During the training:

1. The model goes through forward passes where the embeddings are computed using the encoder, and the edge labels are predicted by the decoder. (W_1 and W_2 are initialized randomly)
2. The MSE loss is back propagated, updating the weight matrices W_1 and W_2 through gradient descent.

Note that the embeddings and matrices are trained at the same time.

Strength

- GraphSAGE explicitly models the connections between users and movies. This means that not only is the direct interaction between a user and a movie considered, but also how they are connected to other users and movies. A user might be similar to another user because they rated a series of common movies.
- GraphSAGE is an inductive method, meaning that it can generalize to unseen nodes (new users or new movies). It learns a function that can generate embeddings for nodes (users or movies) based on their features and graph structure, rather than relying on a fixed interaction matrix.
- Graph-based methods can easily integrate side information such as user profiles, movie metadata (genre, year, director, etc.), and other node attributes. This is done by attaching feature vectors to nodes (e.g., using movie embeddings from metadata in your case).

Limitations

- GraphSAGE is more complex and requires more computational resources than matrix factorization. Training a graph neural network required GPU acceleration and tuning of more hyperparameters (we used Google Collab with CUDA). Hyperparamers include the length of movie/user embeddings, the aggregation function from SageConv, the dropout rate, the number of epochs and learning rate, the LeakyReLU slope rate.

Experimentation

The experimentation we performed involves all the steps detailed in the implementation methods to decide on and implement each part of our matrix factorization. Additionally, we had two different ideas for methods to add that we did not end up finishing and scrapped as it seemed impractical, unhelpful, or both.

The first such method was early stopping, to avoid overfitting if we did not notice the loss continuing to decrease. We wanted to factor this in using test loss rather than training, and to do so would require recalculating E every epoch, so we felt this would be too costly in terms of time and calculation, and thus impractical.

The second method was to incorporate more features/data into our matrix factorization method. We first got movie embeddings using a set of IMDB metadata, as mentioned above. We used this to initialize our V matrix, then paired these with simple user embeddings, which we created by averaging each user's reviews and creating a vector for each one of them to give us an initialization point of our U vector. After some initial testing we realized that this not only greatly increased the time needed to train (as k = 946 to match the length of the word embeddings), but the RMSE dropped substantially as well. We then scrapped this idea too.

Hyperparameter Tuning

We began by initializing our hyperparameters more or less arbitrarily, starting with: k = 10, alpha (learning rate) = 0.01, number of epochs = 500.

We found that hyperparameters are very important and often directly related to the convergence (or lack thereof) of our method. Some preliminary tests showed an alpha of around .001 performed best; anything larger than .01 and smaller than 1e-5 would not work. We also realized that the choice of the number of epochs would be very important to avoid underfitting/overfitting. This differs based on different parameters as well, namely k and alpha, but ~500 epochs seemed to work well. Finally, our testing at first showed that a small k, ~5-15, provided the lowest RMSE. This seemed contrary to what we had heard in class, however.

For the most part, our testing involves nested for loops to test every possible combination of these hyperparameters. Though time consuming this proved effective. As we added more parts into our algorithm, the following new hyperparameters were introduced, which we set to the following values:

L2 mu (regularization term for U) = 0.1, L2 lambda (regularization term for V) = 0.1, momentum = 0.9.

For these values, we found that setting mu and lambda both equal to 0.01 gave us the best results, and a very low momentum factor of 0.1 minimized our RMSE. As a result, we found through more testing that the ideal k value and learning rate value shifted to 160 and 8e-4, respectively, which were more reasonable values. The ideal number of epochs for all these different values came out to be around the 375 range.

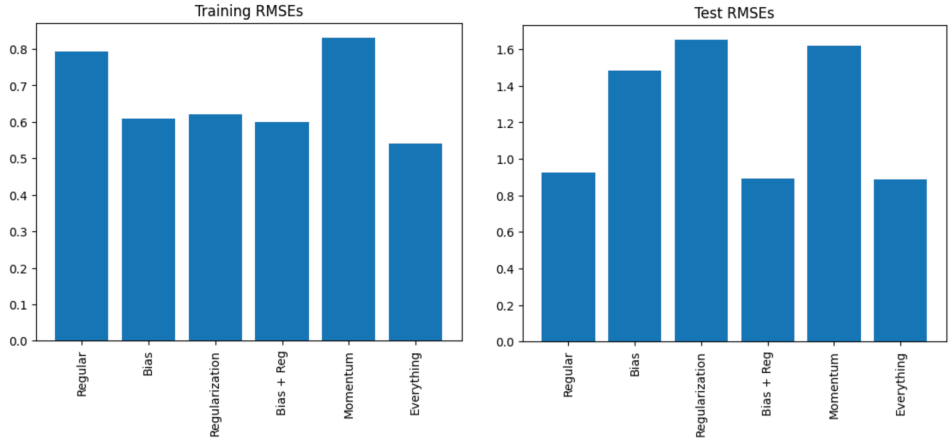
Incorporating different features

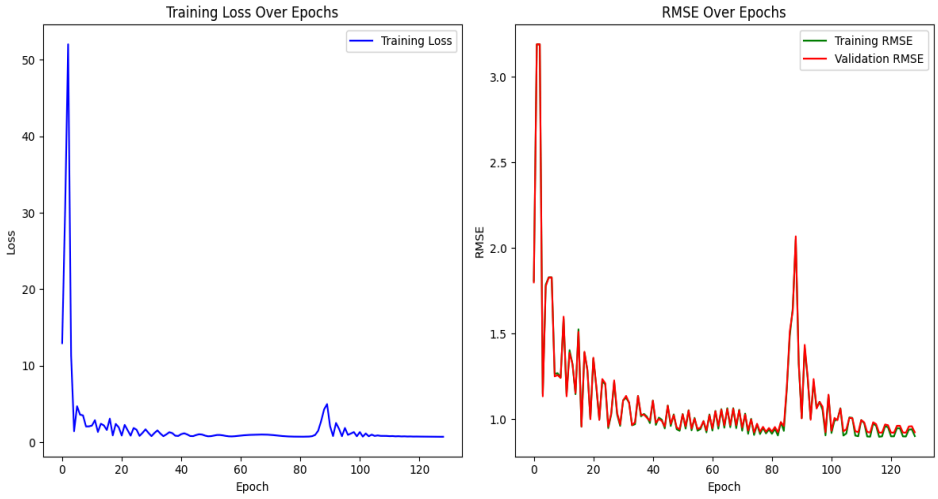
Our matrix factorization method did not incorporate many different features, except for one scrapped attempt to incorporate embeddings for the users and movies into the gradient descent process. As these embeddings would be 'erased' anyways by gradient descent, and as we remarked, significantly worse RMSE and loss values, we decided to move past this idea.

Our graph method allows us to incorporate many different features, mainly related to IMDB metadata about the different movies. We recovered most of this using the IMDB API, which automatically retrieved the following features that we used: Title, Year, Rated, Genre, imdbRating, imdbVotes, Language, Country, Plot, Runtime. As previously mentioned, we used these as features in our movie nodes, which are obviously well incorporated as they are very key to the whole prediction process.

Results

We've obtained the following RMSE values for the matrix factorization approaches:





Conclusion and Learnings

All in all, we found that our more sophisticated graph method, which is able to take into account many different features, performs better than matrix factorization. The latter struggles with RMSE but also especially with accuracy, even when adding methods to try to round and fit the data. This may be because matrix factorization assumes that latent factors are sufficient to represent user preferences and movie characteristics based solely on the user-movie interaction matrix, while GraphSAGE takes into account the graph structure and local neighborhoods, suggesting that users who interact with similar items should have similar embeddings, leveraging the connections beyond direct interactions.

We learned the importance of hyperparameter tuning, how it changes constantly, and can be very time-consuming. Furthermore, it varies per dataset and thus one's validation dataset can yield different results than the true test dataset. The tuning process should be ongoing and is sure to be an extensive, lengthy process.

We also learned the lesson of overfitting, specifically in designing new methods and parts of our algorithm. Using a complex, difficult technique will not necessarily yield better results, especially with such sparse data; furthermore, these methods may do more harm than good when used in conjunction with one another, or what seemed to be a common trend, may cause our metrics to perform the same or slightly worse.

Though we are still convinced that using extensive data is the way to go, with such problems where data is so scarce and we were scrounging for different ways to incorporate more features, going to different lengths with word embeddings, an IMDB API to gain more information, etc. we found that these efforts greatly increase complexity and are perhaps better reserved for a longer-term project. All these were necessary steps for the graph method but proved to be time-consuming to figure out and use. With further experimentation and time to develop these methods, especially the graph method, we think we could get even more promising results.