**Introduction to Recommender Systems (Spring 2023)**

**Homework #2 (100 Pts, April 12)**

**Student ID 2022712151**

**Name 정지원**

**Instruction: You should submit your code and report to i-campus. Follow the submission format below.**

* **RS\_HW2\_STUDENT\_ID\_NAME.zip: Compress 1) your code (models/ folder and code** **for the model that scores the highest in the kaggle competition) and 2) the document.**

**NOTE 1**: You should write your codes **only in ‘EDIT HERE.’**

**NOTE 2:** You need to install Python, NumPy, Scikit-learn (sklearn), Pandas, tqdm, and Matplotlib libraries.

**(1) [40 pts]** We provide datasets and template code in Python. Using a reference code, fill out your code. Run ‘1\_main.py’ to validate your implementation.

**(a)** **[10 pts]** Refer to ‘models/MF\_SGD\_explicit.py’ and write your code to complete the matrix factorization algorithm with modeling user & item bias on ‘models/BiasedMF\_SGD\_explicit.py.’ Initialize all the variables following a normal distribution . The predicted rating of the biased MF is defined as follows:

.

**Note: Fill in your code here. You also have to submit your code to i-campus.**

**Answer:**

|  |
| --- |
| *import* numpy *as* np  *# References : https://yamalab.tistory.com/92*  class BiasedMF\_explicit():  def \_\_init\_\_(self, train, valid, n\_features=20, learning\_rate = 1e-2, reg\_lambda =0.1, num\_epochs = 100):  *self*.train = train  *self*.valid = valid  *self*.num\_users = train.shape[0]  *self*.num\_items = train.shape[1]  *self*.n\_features = n\_features  *self*.num\_epochs = num\_epochs  *self*.learning\_rate = learning\_rate  *self*.reg\_lambda = reg\_lambda    *self*.y = np.zeros\_like(*self*.train)  *for* i, row *in* enumerate(*self*.train):  *self*.y[i, np.where(row > 0.5)[0]] = 1.0  *self*.user\_factors = np.random.normal(scale=1/*self*.n\_features,size=(*self*.num\_users,*self*.n\_features))  *self*.item\_factors = np.random.normal(scale=1/*self*.n\_features,size=(*self*.num\_items,*self*.n\_features))  *# Get the non-zero index of user, item from the ratings matrix*  *self*.user\_indices = np.nonzero(*self*.train)[0]  *self*.item\_indices = np.nonzero(*self*.train)[1]  *self*.num\_ratings = len(*self*.user\_indices)    *# ========================= EDIT HERE ========================*  *# Add bias terms and mu*  *self*.user\_bias = np.zeros(*self*.num\_users)  *self*.item\_bias = np.zeros(*self*.num\_items)    *# mu is mean rating : global average for all ratings*  *self*.mu = np.mean(*self*.train[np.where(*self*.train!=0)])  *# ========================= EDIT HERE ========================*  def mse\_loss(self, y, target, predict):  *return* (y \* (target - predict) \*\* 2).sum()  def fit(self):  ratings = *self*.train  weights = *self*.y    print(f"> Training BiasedMF with SGD for {*self*.num\_epochs} epochs")  *for* epoch *in* range(*self*.num\_epochs):    *# Shuffle the data*  indices = np.random.permutation(*self*.num\_ratings)  *# For each observed entries*  *for* idx *in* indices:  *# Get the user and item index*  user\_id = *self*.user\_indices[idx]  item\_id = *self*.item\_indices[idx]  *# ========================= EDIT HERE ========================*  *# Compute the errors (Use the predict\_single\_entry function)*  error = *self*.train[user\_id,item\_id] - *self*.predict\_single\_entry(user\_id,item\_id)  *## 이곳이 문제였던ㄱ? y야 train이야*  *# Update biases*  *self*.user\_bias[user\_id] += *self*.learning\_rate \* (error - *self*.reg\_lambda \* *self*.user\_bias[user\_id])  *self*.item\_bias[item\_id] += *self*.learning\_rate \* (error - *self*.reg\_lambda \* *self*.item\_bias[item\_id])  *# Update the factors*  tmp = *self*.item\_factors[item\_id,:].copy()  *self*.user\_factors[user\_id,:] += *self*.learning\_rate \* (error \* *self*.item\_factors[item\_id,:] - *self*.reg\_lambda \* *self*.user\_factors[user\_id,:])  *self*.item\_factors[item\_id,:] += *self*.learning\_rate \* (error \* *self*.user\_factors[user\_id,:] - *self*.reg\_lambda \* *self*.item\_factors[item\_id,:])  *self*.item\_factors[item\_id,:] = tmp  *# ========================= EDIT HERE ========================*  *# ========================= EDIT HERE ========================*  *# Compute the loss (Use the predict\_matrix function)*  loss = *self*.mse\_loss(weights,ratings,*self*.predict\_matrix())  *# ========================= EDIT HERE ========================*  *if* epoch % 10 == 0:  print(f"epoch {epoch}, loss: {loss}")  *self*.reconstructed = *self*.predict\_matrix()  def predict\_single\_entry(self, user\_id, item\_id):  predicted = None  *# ========================= EDIT HERE ========================*  predicted = *self*.mu + *self*.user\_bias[user\_id] + *self*.item\_bias[item\_id] + np.dot(*self*.user\_factors[user\_id,:],*self*.item\_factors[item\_id,:].T)  *# ========================= EDIT HERE ========================*  *return* predicted  def predict\_matrix(self):  reconstructed = None  *# ========================= EDIT HERE ========================*  reconstructed = *self*.mu + *self*.user\_bias[:,np.newaxis] + *self*.item\_bias[np.newaxis:,] + np.dot(*self*.user\_factors,*self*.item\_factors.T)    *# ========================= EDIT HERE ========================*  *return* reconstructed  def predict(self, user\_id, item\_ids):  *return* *self*.reconstructed[user\_id, item\_ids] |

**(b)** **[10 pts]** Refer to ‘models/MF\_SGD\_explicit.py’ and write your code to implement the SVD++ algorithm on ‘models/SVDpp\_SGD\_explicit.py’. Initialize all the variables following normal distribution . The predicted rating of the SVD++ is defined as follows:

**Note: Fill in your code here. You also have to submit your code to i-campus.**

**Answer:**

|  |
| --- |
| *import* numpy *as* np  *from* tqdm *import* tqdm  class SVDpp\_explicit():  def \_\_init\_\_(self, train, valid, n\_features=20, learning\_rate = 1e-2, reg\_lambda =0.1, num\_epochs = 100):  *self*.train = train  *self*.valid = valid  *self*.num\_users = train.shape[0]  *self*.num\_items = train.shape[1]  *self*.n\_features = n\_features  *self*.num\_epochs = num\_epochs  *self*.learning\_rate = learning\_rate  *self*.reg\_lambda = reg\_lambda    *self*.y = np.zeros\_like(*self*.train)  *for* i, row *in* enumerate(*self*.train):  *self*.y[i, np.where(row > 0.5)[0]] = 1.0    *# ========================= EDIT HERE ========================*  *# Get the non-zero index of user, item from the ratings matrix*  *self*.user\_indices, *self*.item\_indices = np.nonzero(*self*.train)  *self*.num\_ratings = len(*self*.user\_indices)  *# Add bias terms*  *self*.user\_bias = np.zeros(*self*.num\_users)  *self*.item\_bias = np.zeros(*self*.num\_items)  *self*.bias = np.mean(*self*.train[np.where(*self*.train!=0)])    *self*.user\_factors = np.random.normal(scale=1/*self*.n\_features,size=(*self*.num\_users,*self*.n\_features))  *self*.item\_factors = np.random.normal(scale=1/*self*.n\_features,size=(*self*.num\_items,*self*.n\_features))  *self*.item\_factors\_y = np.random.normal(scale=1/*self*.n\_features,size=(*self*.num\_items,*self*.n\_features))  *# ========================= EDIT HERE ========================*  def mse\_loss(self, y, target, predict):  *return* (y \* (target - predict) \*\* 2).sum()  def fit(self):  ratings = *self*.train  weights = *self*.y    print(f"> Training SVD++ with SGD for {*self*.num\_epochs} epochs")  *# ========================= EDIT HERE ========================*  *# Get the implicit ratings*  *self*.item\_weights = np.zeros(*self*.num\_items)  *for* i *in* range(*self*.num\_items):  *self*.item\_weights[i] = np.sqrt(np.sum(*self*.y[:,i]))      *# ========================= EDIT HERE ========================*  *for* epoch *in* tqdm(range(*self*.num\_epochs), dynamic\_ncols=True):  *# Shuffle the data*  indices = np.random.permutation(*self*.num\_ratings)  *# For each observed entries*  *for* idx *in* indices:    *# Get the user and item index*  user\_id = *self*.user\_indices[idx]  item\_id = *self*.item\_indices[idx]  *# ========================= EDIT HERE ========================*  *# Compute the errors (Use the predict\_single\_entry function)*  error = *self*.train[user\_id,item\_id] - *self*.predict\_single\_entry(user\_id, item\_id)    *# Update biases*  *self*.user\_bias[user\_id] += *self*.learning\_rate \* (error-*self*.reg\_lambda \* *self*.user\_bias[user\_id])  *self*.item\_bias[item\_id] += *self*.learning\_rate \* (error-*self*.reg\_lambda \* *self*.item\_bias[item\_id])      *# Update the factors*  tmp = *self*.item\_factors[item\_id,:].copy()  *self*.item\_factors[item\_id,:] += *self*.learning\_rate \* (error \* *self*.user\_factors[user\_id,:] - *self*.reg\_lambda \* *self*.item\_factors[item\_id,:])  *self*.user\_factors[user\_id,:] += *self*.learning\_rate \* (error \* (tmp + *self*.item\_factors\_y[item\_id,:]) / *self*.item\_weights[item\_id] - *self*.reg\_lambda \* *self*.user\_factors[user\_id,:])  *self*.item\_factors\_y[item\_id,:] += *self*.learning\_rate \*(error \* tmp / *self*.item\_weights[item\_id] - *self*.reg\_lambda \* *self*.item\_factors\_y[item\_id,:])  *# ========================= EDIT HERE ========================*    *# ========================= EDIT HERE ========================*  *# Compute the loss (Use the predict\_matrix function)*  loss = *self*.mse\_loss(weights, ratings, *self*.predict\_matrix())  *# ========================= EDIT HERE ========================*  *if* epoch % 10 == 0:  print(f"epoch {epoch}, loss: {loss}")  *self*.reconstructed = *self*.predict\_matrix()  def predict\_single\_entry(self, user\_id, item\_id):  prediction = None  *# ========================= EDIT HERE ========================*  prediction = *self*.bias + *self*.user\_bias[user\_id] + *self*.item\_bias[item\_id] + np.dot(*self*.user\_factors[user\_id, :], *self*.item\_factors[item\_id, :] + *self*.item\_factors\_y[item\_id, :] / *self*.item\_weights[item\_id])  *# ========================= EDIT HERE ========================*  *return* prediction  def predict\_matrix(self):  reconstructed = None  *# ========================= EDIT HERE ========================*  *# print(self.bias.shape)*  *# reconstructed = self.bias + self.user\_bias[:, np.newaxis] + self.item\_bias[np.newaxis:,] + np.dot(self.user\_factors, (self.item\_factors.T + self.item\_factors\_y.T / self.item\_weights).T)*  reconstructed = *self*.bias + *self*.user\_bias[:,np.newaxis] + *self*.item\_bias[np.newaxis:,] + np.dot(*self*.user\_factors,*self*.item\_factors.T)  *# ========================= EDIT HERE ========================*  *return* reconstructed  def predict(self, user\_id, item\_ids):  *return* *self*.reconstructed[user\_id, item\_ids] |

**(c) [20 pts]** Given the data (‘movielens\_100k.csv’), draw the plot of Recall by **adjusting rank** for MF, Biased MF and SVD++ respectively. By **adjusting dimension sizes(=rank)**, explain the results and how much rank affects RSME. Execute ‘2\_search.py’ to run the code.

**Note: Show your plots and explanations in short (3-5) lines.**

**Answer:**

|  |
| --- |
| **The higher the rank, the more latent factors are used, which can reveal more complex relationships between users and items. In general, the higher the rank, the higher the recall, and after the rmse decreases to a certain point, the performace starts to deteriorate.** |

**(2) [60 pts] (Kaggle competition.)** The task is **top-N recommendation.** Please use the given data to predict top-N recommended movies for each user. Please refer to [**http://www.kaggle.com/competitions/skku-rs2023-assignment-2**](http://www.kaggle.com/competitions/skku-rs2023-assignment-2). It is a private competition, so you should visit via the above link.

**[Notes]**

* “baseline.ipynb” is a baseline code (**most popular method**) for the competition. It includes codes for data preprocessing, model training, evaluation, and making submission file.
* If you cannot access via the link above, try the following link:   
  <https://www.kaggle.com/t/8721cfd6e2f941c180d3536bd9f91d7f>
* Please submit your code for the model that scores the highest in the kaggle competition in ‘**kaggle/**’.

**[Scoring policy]**

The final evaluation will be made by adding the private leaderboard score and the idea score. Please write a report on your project solution with a maximum of 2 pages.

**[Competition Rules]**

* Do not cheat.
* Use Python.
* No limitation on python libraries. (Pytorch, Tensorflow, etc.)
* You must use “**{Student ID}\_{Name}**” for your team name in the Kaggle competition.
* No late submission in the Kaggle competition.
* Any use of external data is prohibited.
* Your submission to Kaggle is limited to five times a day.

**Please write down your solution of Kaggle competition here:**

|  |
| --- |
| **For the Top-N recommendation I used LightGCN and EASE Recommender.**  LightGCN is a graph convolutional network (GCN) that has several advantages over traditional GCNs. Some of the advantages of LightGCN are:   1. Simplicity: LightGCN is a simple and lightweight model that is easy to implement and train. It does not require complex architectures or hyperparameters tuning. 2. Scalability: LightGCN is highly scalable and can handle large-scale graphs with millions of nodes and edges. It achieves this by using a simplified graph convolution operation that reduces the computational complexity. 3. Efficiency: LightGCN is computationally efficient and can be trained on a single GPU. It also has a low memory footprint, which makes it suitable for deployment on resource-constrained devices. 4. Robustness: LightGCN is robust to noise and missing data in the graph. It achieves this by using a graph diffusion process that propagates information across the graph and smooths out noise. 5. Performance: LightGCN achieves state-of-the-art performance on several benchmark datasets for recommendation systems. It outperforms traditional GCNs and other recommendation models in terms of accuracy and efficiency.   I implemented it myself using pytorch and experimented with hyperparameter tuning.  Adamw was used as the optimizer.  Since I implemented it myself without using a module(e.g. surprise,recommenders), I had a lot of trouble improving performance.  The baseline file took a long time to submit, so I modified it a bit. The speed was imporved by using list comprehension without using tqdm.  EASE Recommender proposes a new type of autoencoder that is specifically designed for sparse data.  This model is very simple, so training time and results are fast. |