Recommender System (Spring 2022)

Homework #4 (100 pts, May 11)

Student ID 2019311195

Name 김지유

(1) [75 pts] We provide template code and datasets in Python. Using a reference code ('models/AE_implicit.py'and 'models/MF_implicit.py'), fill out your model code.

(a) [25 pts] Write your code to implement the Collaborative Denoising Autneocder (CDAE) model in 'models/CDAE_implicit.py.'

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

Answer:

```
import numpy as np
import torch
from IPython import embed
from utils import eval_implicit
import os
import math
from time import time
import torch.nn as nn
import torch.nn.functional as F

class CDAE_implicit(torch.nn.Module):
    def __init__(self, train, valid, num_epochs, hidden_dim, learning_rate,
reg_lambda, dropout, device='cpu'):
        super().__init__()
        self.train_mat = train
        self.valid_mat = valid
        self.num_users = train.shape[0]
        self.num_items = train.shape[1]

        self.num_epochs = num_epochs
        self.hidden_dim = hidden_dim
        self.learning_rate = learning_rate
        self.reg_lambda = reg_lambda
        self.dropout = dropout

        self.device = device

        self.build_graph()
```

```python
    def build_graph(self):
        # W, W'와 b, b', V 만들기
        self.enc_w = nn.Parameter(torch.ones(self.num_items, self.hidden_dim))
        self.enc_b = nn.Parameter(torch.ones(self.hidden_dim))
        nn.init.xavier_uniform_(self.enc_w)
        nn.init.normal_(self.enc_b, 0, 0.001)

        self.dec_w = nn.Parameter(torch.ones(self.hidden_dim, self.num_items))
        self.dec_b = nn.Parameter(torch.ones(self.num_items))
        nn.init.xavier_uniform_(self.dec_w)
        nn.init.normal_(self.dec_b, 0, 0.001)

        '''
        Implement user_embedding parameters
        '''
        # ========================= EDIT HERE ========================
        self.user_embedding = nn.Embedding(self.num_users, self.hidden_dim)

        # ========================= EDIT HERE ========================

        # 최적화 방법 설정
        self.optimizer = torch.optim.Adam(self.parameters(),
lr=self.learning_rate, weight_decay=self.reg_lambda)

        # 모델을 device로 보냄
        self.to(self.device)

    def forward(self, u, x):

        '''
        Implement forward pass
        '''
        # ========================= EDIT HERE ========================

        # 입력의 일부를 제거
        denoised_x = F.dropout(x, p=self.dropout, training=self.training)

        # encoder 과정
        h = torch.sigmoid(denoised_x @ self.enc_w + self.enc_b +
self.user_embedding(u))

        # decoder 과정
        output = torch.sigmoid(h @ self.dec_w + self.dec_b)

```

```
        # ========================= EDIT HERE ========================

        return output

    def fit(self):
        train_matrix = torch.FloatTensor(self.train_mat).to(self.device)
        user_idx = np.arange(self.num_users)
        user_idx = torch.LongTensor(user_idx).to(self.device)

        for epoch in range(0, self.num_epochs):
            self.train()
            loss = self.train_model_per_batch(user_idx, train_matrix)

            if torch.isnan(loss):
                print('Loss NAN. Train finish.')
                break

            if epoch % 20 == 0:
                with torch.no_grad():
                    self.eval()
                    self.reconstructed = self.forward(user_idx,
train_matrix).detach().cpu().numpy()

                    top_k=50
                    print("[CDAE] epoch %d, loss: %f"%(epoch, loss))
                    prec, recall, ndcg = eval_implicit(self, self.train_mat,
self.valid_mat, top_k)
                    print(f"(CDAE VALID) prec@{top_k} {prec}, recall@{top_k}
{recall}, ndcg@{top_k} {ndcg}")
                    self.train()

    def train_model_per_batch(self, user_idx, train_matrix):
        # grad 초기화
        self.optimizer.zero_grad()

        # 모델 forwrad
        output = self.forward(user_idx, train_matrix)

        # loss 구함
        loss = F.binary_cross_entropy(output, train_matrix,
reduction='none').sum(1).mean()

        # 역전파
        loss.backward()

        # 최적화
        self.optimizer.step()
```

```
        return loss

    def predict(self, user_id, item_ids):
        return self.reconstructed[user_id, item_ids]


```

(b) [25 pts] Write your code to implement the Multinomial Variational Autneocder (MultVAE) model in 'models/ MultVAE_implicit.py.'

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

Answer:

```
import numpy as np
import torch
from IPython import embed
from utils import eval_implicit
import os
import math
from time import time
import torch.nn as nn
import torch.nn.functional as F

class MultVAE_implicit(torch.nn.Module):
    def __init__(self, train, valid, num_epochs, hidden_dim, learning_rate,
reg_lambda, dropout, device='cpu'):
        super().__init__()
        self.train_mat = train
        self.valid_mat = valid
        self.num_users = train.shape[0]
        self.num_items = train.shape[1]

        self.num_epochs = num_epochs
        self.hidden_dim = hidden_dim
        self.learning_rate = learning_rate
        self.reg_lambda = reg_lambda

        self.total_anneal_steps = 200000
        self.anneal_cap = 0.2

        self.dropout = dropout

```

```
        self.update_count = 0
        self.device = device

        self.build_graph()

```

```
    def build_graph(self):
        # W, W'와 b, b'만들기
        self.enc_w = nn.Parameter(torch.ones(self.num_items, self.hidden_dim * 2))
        self.enc_b = nn.Parameter(torch.ones(self.hidden_dim * 2))
        nn.init.xavier_uniform_(self.enc_w)
        nn.init.normal_(self.enc_b, 0, 0.001)

        self.dec_w = nn.Parameter(torch.ones(self.hidden_dim, self.num_items))
        self.dec_b = nn.Parameter(torch.ones(self.num_items))
        nn.init.xavier_uniform_(self.dec_w)
        nn.init.normal_(self.dec_b, 0, 0.001)

        # 최적화 방법 설정
        self.optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate, weight_decay=self.reg_lambda)

        # 모델을 device로 보냄
        self.to(self.device)

```

```
    def forward(self, x):
        '''
        Implement forward pass
        '''
        # ========================= EDIT HERE ========================

        # 입력의 일부를 제거
        denoised_x =  F.dropout(F.normalize(x), p=self.dropout, training=self.training)

        # encoder 과정
        h =  torch.sigmoid(denoised_x @ self.enc_w + self.enc_b)

        # 잠재인수 z 만들기
        mu_q =  h[:, :self.hidden_dim]
        logvar_q =  h[:, self.hidden_dim:]
        std_q =  torch.exp(0.5 * logvar_q)

        epsilon = torch.zeros_like(std_q).normal_(mean=0, std=0.01) # Don't edit here
        sampled_z =  mu_q + self.training * epsilon * std_q

        # decoder 과정
```

```
        output =  torch.sigmoid(sampled_z @ self.dec_w + self.dec_b)

        # KL loss
        if self.training:
            kl_loss =  ((0.5 * (-logvar_q + torch.exp(logvar_q) + torch.pow(mu_q,
2) - 1)).sum(1)).mean()
            return output, kl_loss
        else:
            return output

        # ========================= EDIT HERE ========================

```

```
    def fit(self):
        train_matrix = torch.FloatTensor(self.train_mat).to(self.device)

        for epoch in range(0, self.num_epochs):
            self.train()

            if self.total_anneal_steps > 0:
                self.anneal = min(self.anneal_cap, 1. * self.update_count /
self.total_anneal_steps)
            else:
                self.anneal = self.anneal_cap

            loss = self.train_model_per_batch(train_matrix)

            if torch.isnan(loss):
                print('Loss NAN. Train finish.')
                break

            if epoch % 20 == 0:
                with torch.no_grad():
                    self.eval()
                    self.reconstructed =
self.forward(train_matrix).detach().cpu().numpy()

                    top_k=50
                    print("[MultVAE CF] epoch %d, loss: %f"%(epoch, loss))
                    prec, recall, ndcg = eval_implicit(self, self.train_mat,
self.valid_mat, top_k)
                    print(f"(MultVAE VALID) prec@{top_k} {prec}, recall@{top_k}
{recall}, ndcg@{top_k} {ndcg}")
                    self.train()

```

```
    def train_model_per_batch(self, train_matrix):
        # grad 초기화
        self.optimizer.zero_grad()
```

```
        # 모델 forwrad
        output, kl_loss = self.forward(train_matrix)

        '''
        Implement multinomial likelihood
        '''
        # ========================= EDIT HERE ========================
        multinomial = -torch.mean(torch.sum(F.log_softmax(output, 1) *
train_matrix, -1))
        # ========================= EDIT HERE ========================

        loss = multinomial + kl_loss * self.anneal

        # 역전파
        loss.backward()

        # 최적화
        self.optimizer.step()

        self.update_count += 1

        return loss

    def predict(self, user_id, item_ids):
        return self.reconstructed[user_id, item_ids]
```

(c) [25 pts] Write your code to implement the LightGCN model in 'models/LightGCN_implicit.py.'

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

Answer:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from time import time
import numpy as np
import scipy.sparse as sp
from IPython import embed
from utils import eval_implicit

class LightGCN_implicit(nn.Module):
    def __init__(self, train, valid, learning_rate, regs, batch_size, num_epochs,
emb_size, num_layers, node_dropout, device='cpu'):
```

```
        super(LightGCN_implicit, self).__init__()

        self.train_data = train
        self.valid_data = valid

        self.train_mat = sp.csr_matrix(train)
        self.valid_mat = sp.csr_matrix(valid)

        self.num_users, self.num_items = self.train_mat.shape

        self.R = sp.csr_matrix(train)

        self.norm_adj = self.create_adj_mat()

        self.learning_rate = learning_rate
        self.device = device
        self.emb_size = emb_size
        self.num_layers = num_layers
        self.batch_size = batch_size
        self.num_epochs = num_epochs
        self.node_dropout = node_dropout

        self.decay = regs

        """
        *********************************************************
        Init the weight of user-item.
        """
        self.embedding_dict = self.init_weight()

        """
        *********************************************************
        Get sparse adj.
        """
        self.optimizer = optim.Adam(self.parameters(), lr=self.learning_rate)
        self.sparse_norm_adj =
self._convert_sp_mat_to_sp_tensor(self.norm_adj).to(self.device)
        self.to(self.device)

    def init_weight(self):
        # xavier init
        initializer = nn.init.xavier_uniform_

        embedding_dict = nn.ParameterDict({
            'user_emb': nn.Parameter(initializer(torch.empty(self.num_users,
                                                 self.emb_size))),
            'item_emb': nn.Parameter(initializer(torch.empty(self.num_items,
                                                 self.emb_size)))
```

```python
        })

        return embedding_dict

    def _convert_sp_mat_to_sp_tensor(self, X):
        coo = X.tocoo()
        i = torch.LongTensor([coo.row, coo.col])
        v = torch.from_numpy(coo.data).float()
        return torch.sparse.FloatTensor(i, v, coo.shape)

    def sparse_dropout(self, x, rate, noise_shape):
        random_tensor = 1 - rate
        random_tensor += torch.rand(noise_shape).to(x.device)
        dropout_mask = torch.floor(random_tensor).type(torch.bool)
        i = x._indices()
        v = x._values()

        i = i[:, dropout_mask]
        v = v[dropout_mask]

        out = torch.sparse.FloatTensor(i, v, x.shape).to(x.device)

        return out * (1. / (1 - rate))

    def rating(self, u_g_embeddings, pos_i_g_embeddings):
        return torch.matmul(u_g_embeddings, pos_i_g_embeddings.t())

    def forward(self, users, pos_items, neg_items, drop_flag=False):
        # 사용자-항목 상호작용 그래프 정점 드롭아웃
        A_hat = self.sparse_dropout(self.sparse_norm_adj,
                                    self.node_dropout,
                                    self.sparse_norm_adj._nnz()) if drop_flag else
self.sparse_norm_adj

        # 초기 임베딩 불러오기
        ego_embeddings = torch.cat([self.embedding_dict['user_emb'],
                                    self.embedding_dict['item_emb']], 0)

        # 각각의 레이어에서의 임베딩 결과를 저장하기 위한 공간 생성
        all_embeddings = [ego_embeddings]

```

```python
        '''
        Implement GCN process
        '''
        # ========================= EDIT HERE ========================
        # 레이어마다 GCN 수행
        for k in range(self.num_layers):
            # 메시지 통합 (Message Aggregation)
```

```
            norm_embeddings = torch.sparse.mm(A_hat, ego_embeddings)

            # k번째 임베딩 저장
            #all_embeddings +=  norm_embeddings
            all_embeddings.append(norm_embeddings)
        # ========================= EDIT HERE ========================

```

```
        # 동일 가중치 합으로 최종 임베딩 생성
        all_embeddings = torch.stack(all_embeddings, 1)
        final_embeddings = torch.mean(all_embeddings, 1)

        u_g_embeddings = final_embeddings[:self.num_users, :] # u_embedding
        i_g_embeddings = final_embeddings[self.num_users:, :] # i_embedding

        # 필요한 임베딩 가져가기
        u_g_embeddings = u_g_embeddings[users, :] # user embedding
        pos_i_g_embeddings = i_g_embeddings[pos_items, :] # positive item
embedding
        neg_i_g_embeddings = i_g_embeddings[neg_items, :] # negative item
embedding

        return u_g_embeddings, pos_i_g_embeddings, neg_i_g_embeddings,
i_g_embeddings

    def fit(self):
        user_idx = np.arange(self.num_users)

        for epoch in range(self.num_epochs):
            epoch_loss = 0.0

            self.train()

            np.random.RandomState(12345).shuffle(user_idx)

            batch_num = int(len(user_idx) / self.batch_size) + 1

            for batch_idx in range(batch_num):
                batch_users =
user_idx[batch_idx*self.batch_size:(batch_idx+1)*self.batch_size]
                batch_matrix =
torch.FloatTensor(self.train_mat[batch_users, :].toarray()).to(self.device)
                batch_users = torch.LongTensor(batch_users).to(self.device)
                batch_loss = self.train_model_per_batch(batch_matrix, batch_users)

                if torch.isnan(batch_loss):
                    print('Loss NAN. Train finish.')
                    break

```

```
                epoch_loss += batch_loss


            if epoch % 20 == 0:
                with torch.no_grad():
                    self.eval()

                    top_k=50
                    print("[LightGCN] epoch %d, loss: %f"%(epoch, epoch_loss))

                    prec, recall, ndcg = eval_implicit(self, self.train_data,
self.valid_data, top_k)
                    print(f"(LightGCN VALID) prec@{top_k} {prec}, recall@{top_k}
{recall}, ndcg@{top_k} {ndcg}")
                    self.train()
```

```
    def train_model_per_batch(self, train_matrix, batch_users, pos_items=0,
neg_items=0):
        # grad 초기화
        self.optimizer.zero_grad()

        u_g_embeddings, _, _, i_g_embeddings = self.forward(batch_users, 0, 0)

        '''
        Implement output and loss function (binary CE loss)
        '''
        # ========================= EDIT HERE ========================
        output = self.rating(u_g_embeddings, i_g_embeddings)

        # binary CE loss
        loss = F.binary_cross_entropy(torch.sigmoid(output), train_matrix,
reduction="none").sum(1).mean()
        # ========================= EDIT HERE ========================

        # 역전파
        loss.backward()

        # 최적화
        self.optimizer.step()

        return loss
```

```
    def predict(self, user_ids, item_ids):
        with torch.no_grad():
```

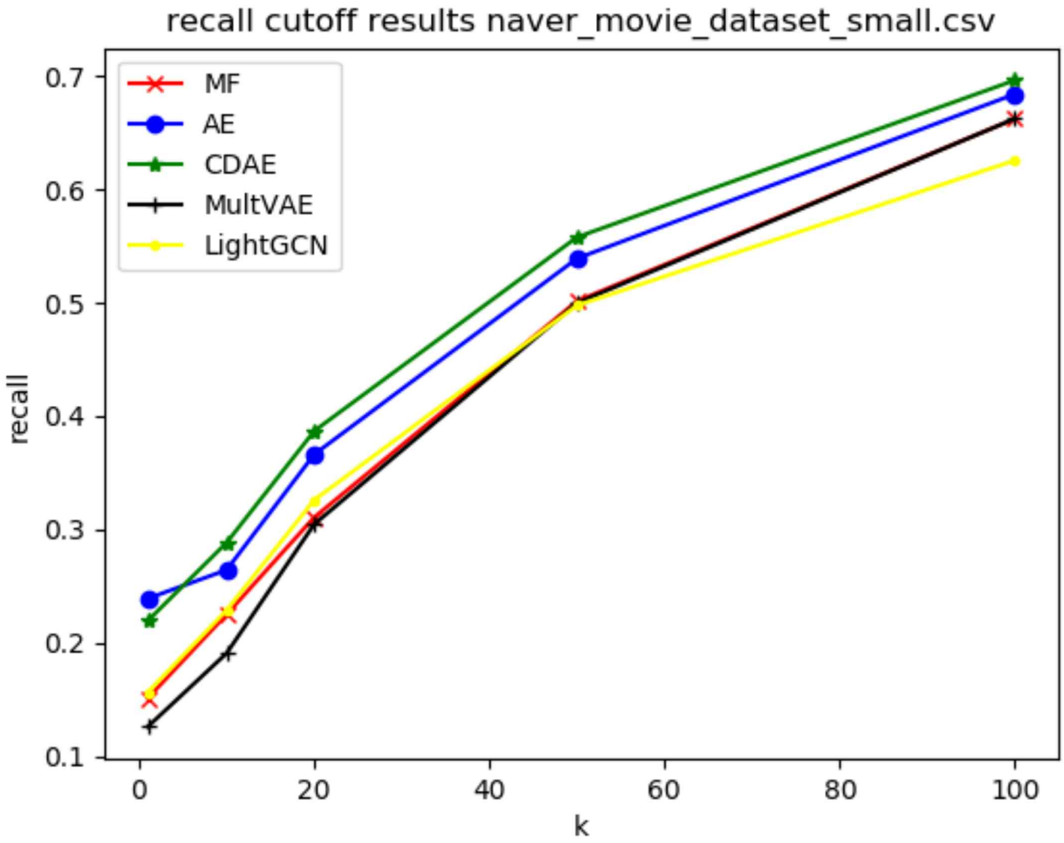
```python
            u_g_embeddings, _, _, i_g_embeddings = self.forward(user_ids, 0, 0)

            '''
            Implement output
            '''
            # ========================= EDIT HERE ========================
            output =  self.rating(u_g_embeddings, i_g_embeddings)
            # ========================= EDIT HERE ========================

            predict_ = output.detach().cpu().numpy()
            return predict_[item_ids]

```

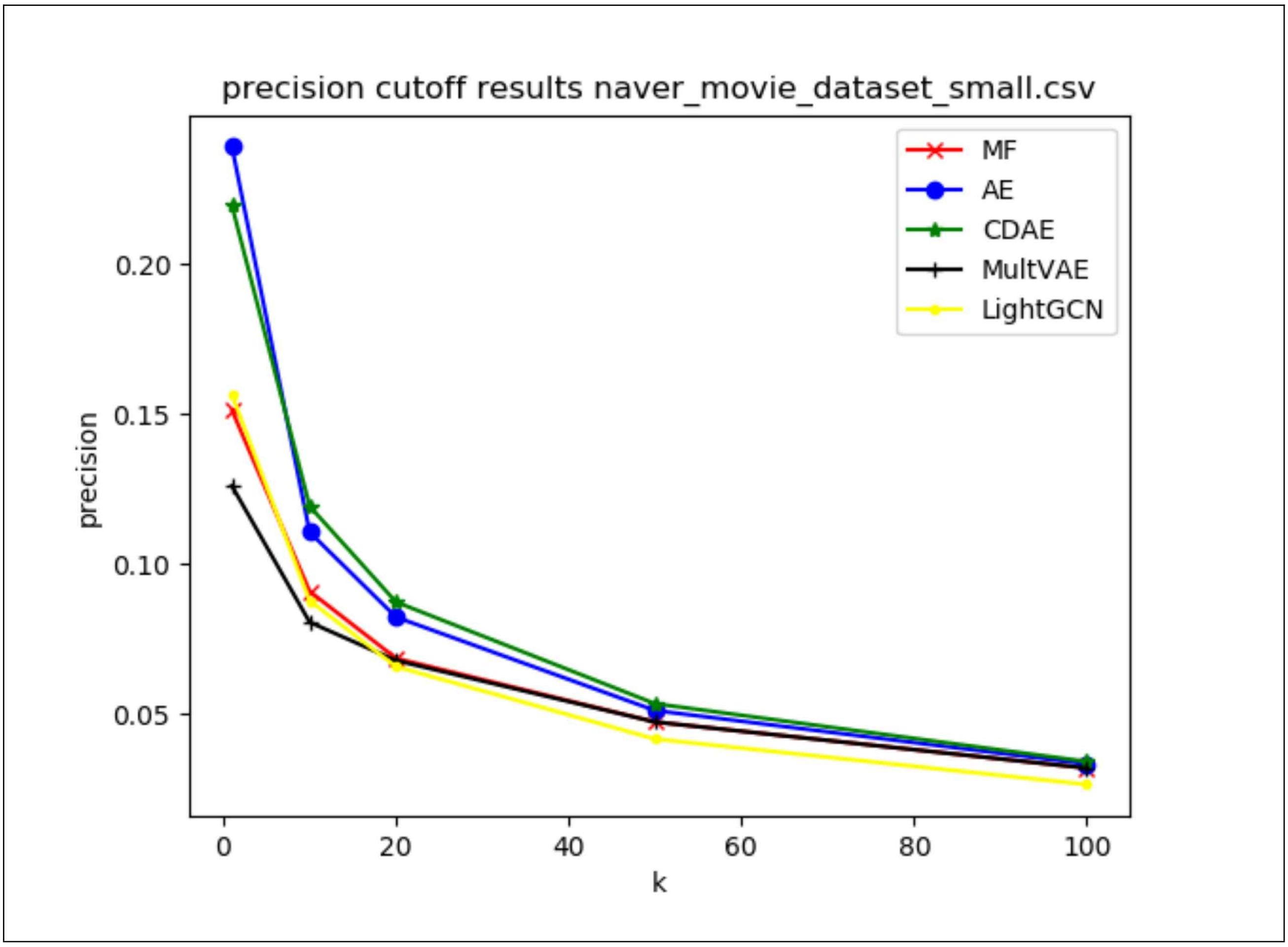
```python
    def create_adj_mat(self):
        adj_mat = sp.dok_matrix((self.num_users + self.num_items, self.num_users +
self.num_items), dtype=np.float32)

        adj_mat = adj_mat.tolil()
        R = sp.csr_matrix(self.R).tolil()

        adj_mat[:self.num_users, self.num_users:] = R
        adj_mat[self.num_users:, :self.num_users] = R.T
        adj_mat = adj_mat.todok()

        # D^-1/2 * A * D^-1/2
        rowsum = np.array(adj_mat.sum(axis=1))

        d_inv = np.power(rowsum, -0.5).flatten()
        d_inv[np.isinf(d_inv)] = 0.
        d_mat = sp.diags(d_inv)

        norm_adj = d_mat.dot(adj_mat).dot(d_mat)
        norm_adj = norm_adj.tocsr()

        return norm_adj

    def _convert_sp_mat_to_sp_tensor(self, X):
        coo = X.tocoo().astype(np.float32)
        row = torch.Tensor(coo.row).long()
        col = torch.Tensor(coo.col).long()
        index = torch.stack([row, col])
        data = torch.FloatTensor(coo.data)

        return torch.sparse.FloatTensor(index, data, torch.Size(coo.shape))
```

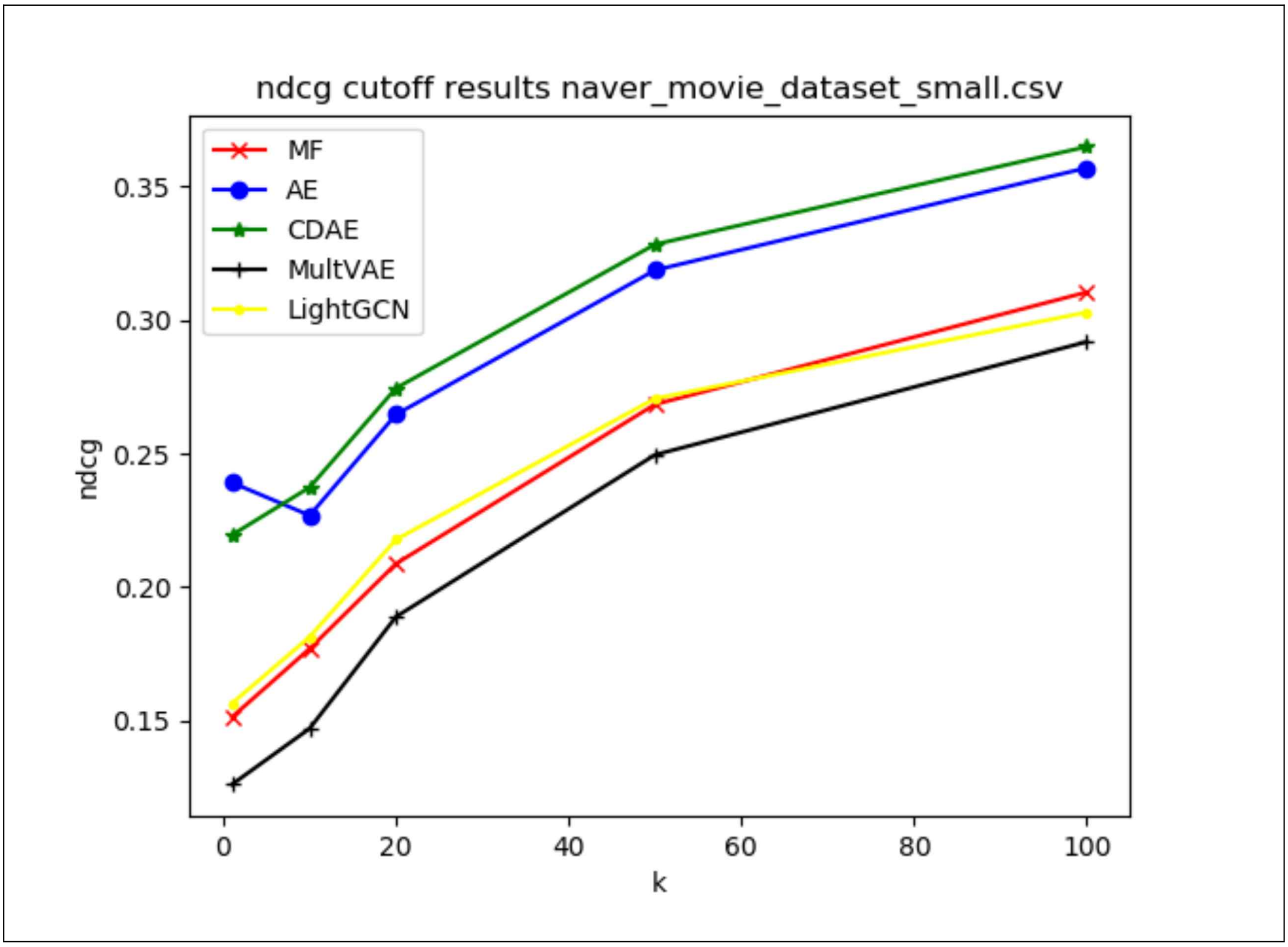
(2) [25 pts] Given the data ('naver_movie_dataset_100k.csv' and 'movielens_100k.csv'), draw the plots of Precision, Recall, and NDCG by adjusting the cutoff at MF, AE, CDAE, MultVAE, and LightGCN. Explain the results by changing cutoff sizes (number of recommended items). Run '1_cutoff.py' to run the code.

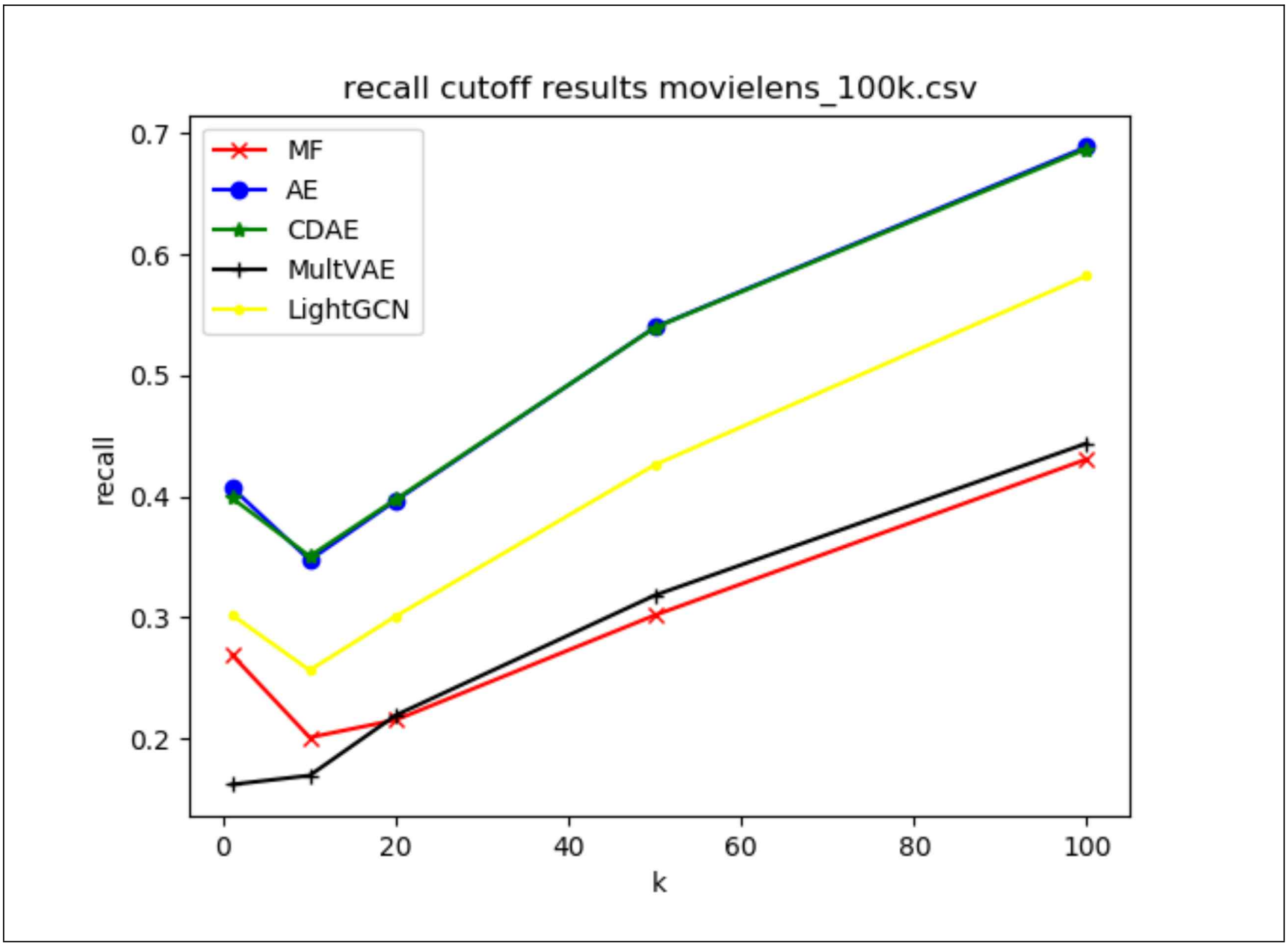
Note: Please show your plots over two datasets and explain the results.
Answer:

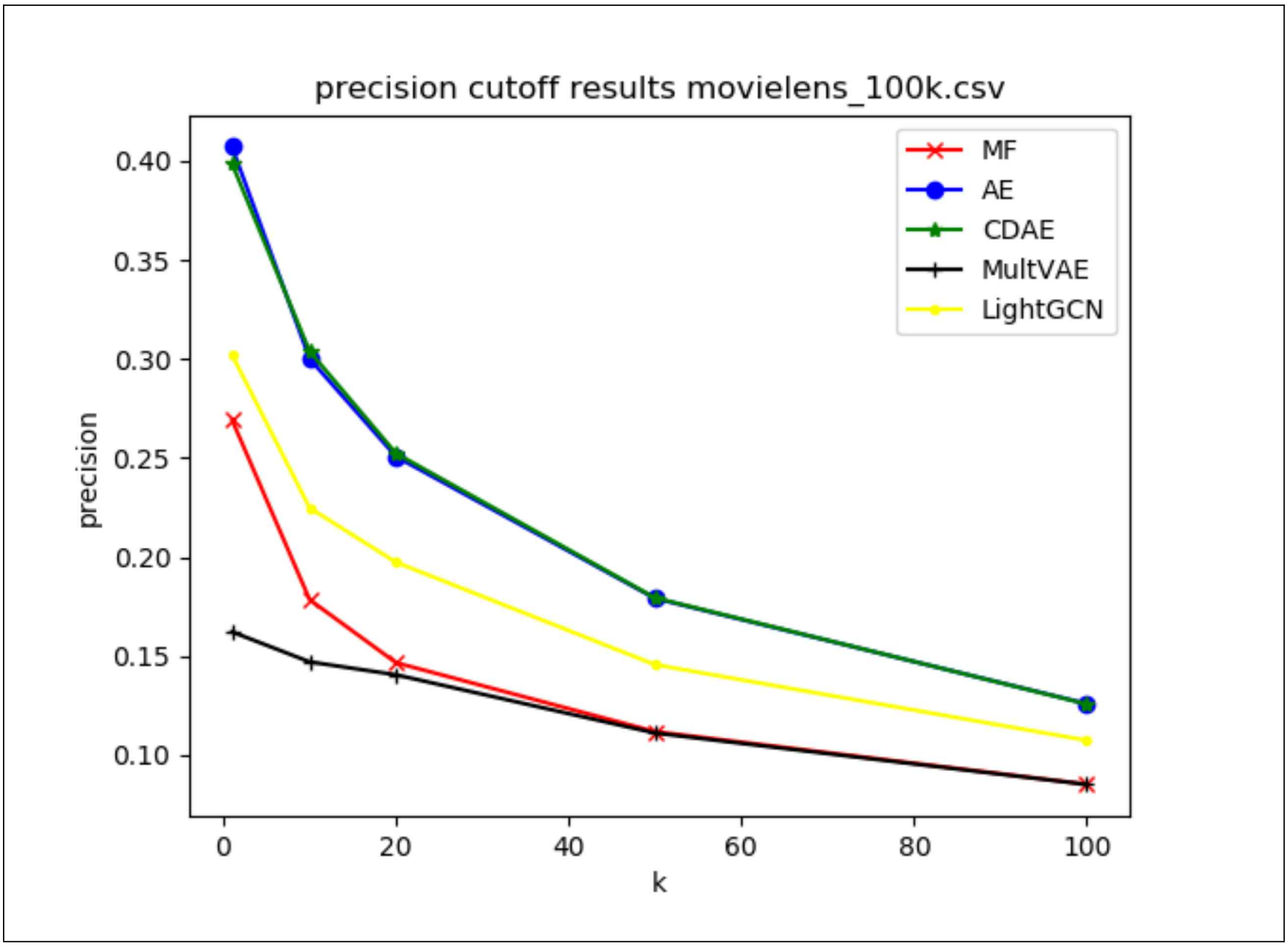
Naver_movie_dataset.csv와 movielens_100k.csv에 데이터에 대해 모든 모델이 cutoff가 증가함에 따라 recall, ndcg는 계속 증가하였고, precision은 감소하였다.

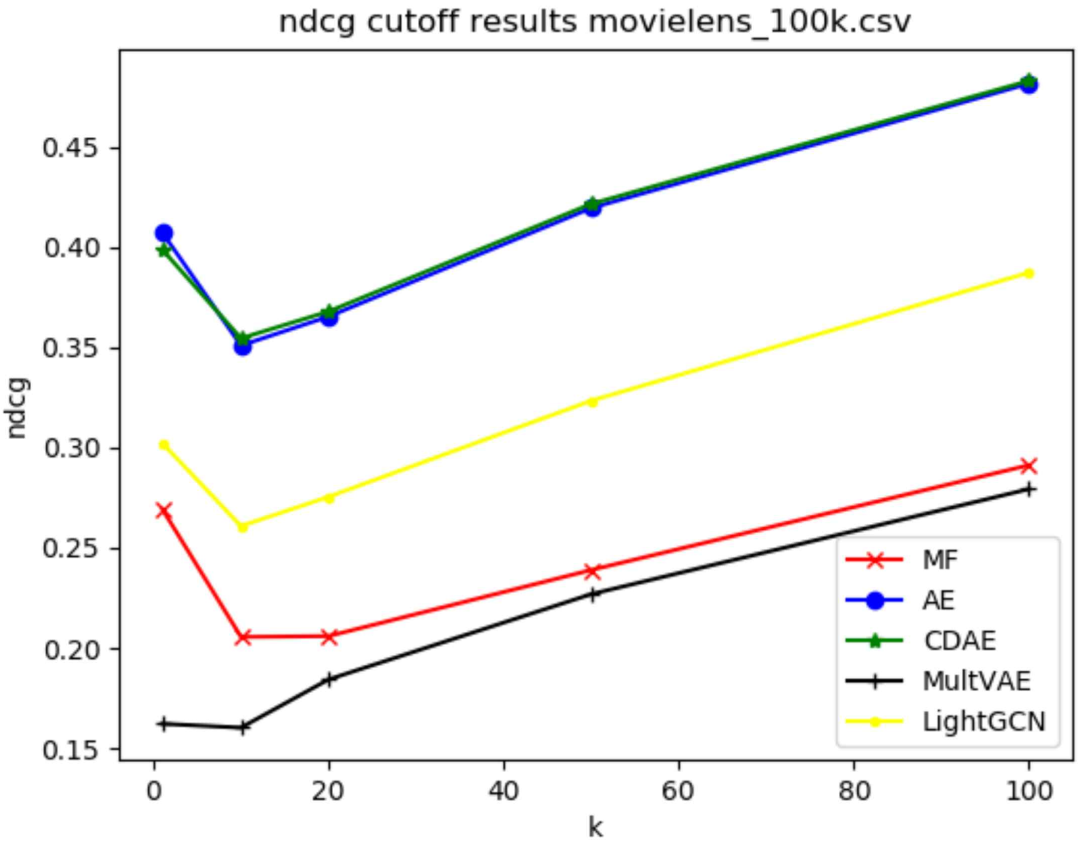



precision cutoff results naver_movie_dataset_small.csv
precision
k
MF
AE
CDAE
MultVAE
LightGCN
0.20
0.15
0.10
0.05
0
20
40
60
80
100


ndcg cutoff results naver_movie_dataset_small.csv
MF
AE
CDAE
MultVAE
LightGCN
ndcg
0.15
0.20
0.25
0.30
0.35
0
20
40
60
80
100
k


recall cutoff results movielens_100k.csv
MF
AE
CDAE
MultVAE
LightGCN
recall
k
0.2
0.3
0.4
0.5
0.6
0.7
0
20
40
60
80
100


precision cutoff results movielens_100k.csv
MF
AE
CDAE
MultVAE
LightGCN
precision
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0
20
40
60
80
100
k



ndcg cutoff results movielens_100k.csv
ndcg
k
0.15
0.20
0.25
0.30
0.35
0.40
0.45
0
20
40
60
80
100
MF
AE
CDAE
MultVAE
LightGCN