Recommender System (Spring 2022)

Homework #3 (120 pts, April 27)

Student ID **<u>2019311195</u>**

Name **<u>김지유</u>**

(1) [80 pts] We are given template code and datasets in Python. Using a reference code (‘models/MF_implicit.py’and ‘models/WMF_implicit.py’), fill out your model code. Run ‘0_check.py’ and ‘1_main.py’ to validate your implementation.

(a) [20 pts] Write your code to implement the Bayesian Personalized Ranking (BPR) model in ‘models/ BPR_implicit.py’. Initialize all the variables following a normal distribution $\mathcal{N}(0, 0.01)$. The predicted rating of the BPR is defined as follows:

$$\hat{r}_{ui} = u_u v_i^T + b_i \text{ where } b_i \text{ denotes bias for item } i.$$

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

Answer:

```
import numpy as np
import torch
from tqdm import tqdm
from IPython import embed
from utils import eval_implicit

class BPR_implicit_model(torch.nn.Module):
    def __init__(self, num_users, num_items, n_features):
        super().__init__()
        self.user_factors = torch.nn.Embedding(num_users, n_features,
sparse=False)
        self.item_factors = torch.nn.Embedding(num_items, n_features,
sparse=False)
        self.item_bias = torch.nn.Embedding(num_items, 1, sparse=False)

        torch.nn.init.normal_(self.user_factors.weight, std=0.01)
        torch.nn.init.normal_(self.item_factors.weight, std=0.01)
        torch.nn.init.normal_(self.item_bias.weight, std=0.01)

    def forward(self, user_ids, item_ids):
        user_embs = self.user_factors(user_ids)
        item_embs = self.item_factors(item_ids)
```

```
        item_biases = self.item_bias(item_ids).squeeze()

        predictions = torch.sum(user_embs * item_embs, dim=1) + item_biases
        return predictions


class BPR_implicit():
    def __init__(self, train, valid, n_features=20, learning_rate = 1e-2,
reg_lambda =0.1, num_epochs = 100, batch_size=102400, num_negative=3,
device='cpu'):
        self.train = train
        self.valid = valid
        self.num_users = train.shape[0]
        self.num_items = train.shape[1]
        self.num_epcohs = num_epochs
        self.n_features = n_features
        self.batch_size = batch_size
        self.num_negative = num_negative
        self.device = device

        self.model = BPR_implicit_model(self.num_users, self.num_items,
self.n_features).to(device)
        self.BCE_loss = torch.nn.BCEWithLogitsLoss()
        self.optimizer = torch.optim.Adam(self.model.parameters(),
lr=learning_rate, weight_decay=reg_lambda)

    def fit(self):
        # ========================= EDIT HERE ========================
        user_rated_dict = dict()
        user_not_rated_dict = dict()
        for u in range(self.num_users):
            user_rated_dict[u] = np.where(self.train[u, :] > 0)[0]
            user_not_rated_dict[u] = np.setdiff1d(np.arange(self.num_items),
user_rated_dict[u], assume_unique=True)

        for epoch in range(self.num_epcohs):
            train_data = []
            '''
            Implement making training data
            e.g.) train_data = [(user_id, pos_item_id, neg_item_id), ...]
            '''
            # IMPLEMENT HERE
            for u in range(self.num_users):
                user_id = u
                for pos_item_id in user_rated_dict[u]:
                    neg_item_id = np.random.choice(user_not_rated_dict[u],
1).item()
                    while [user_id, pos_item_id, neg_item_id] in train_data:
```

```
                        neg_item_id = np.random.choice(user_not_rated_dict[u],
1).item()
                    train_data.append([user_id, pos_item_id, neg_item_id])

            #print(train_data)
            train_data = torch.tensor(np.array(train_data))

            train_loader = torch.utils.data.DataLoader(train_data,
batch_size=self.batch_size, shuffle=True)
            epoch_loss = 0
            for train in train_loader:
                users = train[:, 0].to(self.device)
                item_is = train[:, 1].to(self.device)
                item_js = train[:, 2].to(self.device)

                '''
                Implement prediction and loss
                '''
                # IMPLEMENT HERE
                prediction_is = self.model.forward(users, item_is)
                prediction_js = self.model.forward(users, item_js)
                loss = -(prediction_is - prediction_js).sigmoid().log().sum()
                #loss = torch.Tensor([0.0]).to(self.device)

                epoch_loss += loss.item() / len(train)

                # gradient reset
                self.optimizer.zero_grad()

                # Backpropagate
                loss.backward()

                # Update the parameters
                self.optimizer.step()

            if epoch % 1 == 0:
                top_k=50
                prec, recall, ndcg, mrr, mAP = eval_implicit(self, self.train,
self.valid, top_k)
                print("[BPR] epoch %d, loss: %f"%(epoch,
epoch_loss/len(train_loader)))
                print(f"(BPR VALID) prec@{top_k} {prec}, recall@{top_k} {recall},
ndcg@{top_k} {ndcg}, mrr@{top_k} {mrr}, map@{top_k} {mAP}")
        # ========================= EDIT HERE ========================

    def predict(self, user_id, item_ids):
        with torch.no_grad():
            user_id = torch.tensor([user_id]).to(self.device)
```

```
        item_ids = torch.tensor(item_ids).to(self.device)
        prediction = self.model.forward(user_id, item_ids)
        return prediction.cpu().numpy()
```

(a-1) [**20 pts]** BPR usually uses random sampling for negative items. Implement a better sampling method than random sampling in ‘BPR_implicit.py’. Explain why your sampling method can be better than random sampling.

Note: Fill in your code here and explain your sampling method. You also have to submit your code to i-campus.

(b) [20 pts] Write your code to implement the Factored Item Similarity Models (FISM) in ‘models/FISM_implicit.py.’ Initialize all the variables following a normal distribution $\mathcal{N}(0, 0.01)$. The predicted rating of the FISM is defined as follows:

$$\hat{r}_{ui} = b_u + b_i + (n_u^+ - 1)^{-\alpha} \sum_{j \in \mathcal{R}_u^+ \setminus \{i\}} \mathrm{P}_j \, \mathrm{Q}_i^T$$

where b_u and b_i denote bias for user u and item i,
$\mathcal{R}_u^+$ denotes the set of items rated by user u,
n_u^+ denotes the number of items rated by user u,
$\alpha\ denotes\ a\ user\ specified\ parameter$respectively.

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

Answer:

```python
import numpy as np
import torch
from tqdm import tqdm
from IPython import embed
from utils import eval_implicit


class FISM_implicit_model(torch.nn.Module):
    def __init__(self, num_items, n_features, alpha):
        super().__init__()
        self.alpha = alpha

        self.item_factors_P = torch.nn.Embedding(num_items, n_features,
sparse=False)
        self.item_factors_Q = torch.nn.Embedding(num_items, n_features,
sparse=False)
        self.item_bias = torch.nn.Embedding(num_items, 1, sparse=False)

        torch.nn.init.normal_(self.item_factors_P.weight, std=0.01)
        torch.nn.init.normal_(self.item_factors_Q.weight, std=0.01)
        torch.nn.init.normal_(self.item_bias.weight, std=0.01)

    def forward(self, user_rating, item_ids, pos_item=False):
        #predictions = torch.tensor(0.0).to(user_rating.device)
        '''
        Implement forward pass
        '''
        # ========================= EDIT HERE ========================
        predictions = torch.matmul(self.item_factors_P(item_ids).sum(axis=0),
self.item_factors_Q.weight.T)
        predictions = predictions * np.power(len(item_ids)-1, -self.alpha)
        predictions = predictions + self.item_bias(item_ids)
        # ========================= EDIT HERE ========================
        return predictions

# It is FISM AUC version
class FISM_implicit():
    def __init__(self, train, valid, n_features=20, learning_rate=1e-2,
reg_lambda=0.1, num_epochs=100,
                alpha=0.5, num_negative=3, batch_size=102400, device='cpu'):
        self.train = train
        self.valid = valid
        self.num_users = train.shape[0]
        self.num_items = train.shape[1]
        self.num_epcohs = num_epochs
        self.n_features = n_features
        self.device = device
```

```python
        self.alpha = alpha
        self.num_negative = num_negative
        self.batch_size = batch_size

        self.model = FISM_implicit_model(self.num_items, self.n_features,
self.alpha).to(device)
        self.optimizer = torch.optim.Adam(self.model.parameters(),
lr=learning_rate, weight_decay=reg_lambda)

    def fit(self):
        # ========================= EDIT HERE ========================
        user_rated_dict = dict()
        user_not_rated_dict = dict()
        for u in range(self.num_users):
            user_rated_dict[u] = np.where(self.train[u, :] > 0)[0]
            user_not_rated_dict[u] = np.setdiff1d(np.arange(self.num_items),
user_rated_dict[u], assume_unique=True)

        for epoch in range(self.num_epcohs):
            train_data = []
            '''
            Implement making training data
            e.g.) train_data = [(user_id, pos_item_id, neg_item_id), ...]
            '''
            # IMPLEMENT HERE
            for user_id in range(self.num_users):
                for pos_item_id in user_rated_dict[u]:
                    for _ in range(self.num_negative):
                        neg_item_id = np.random.choice(user_not_rated_dict[u],
1).item()
                        while [user_id, pos_item_id, neg_item_id] in train_data:
                            neg_item_id = np.random.choice(user_not_rated_dict[u],
1).item()
                        train_data.append([user_id, pos_item_id, neg_item_id])

            train_data = torch.tensor(np.array(train_data))

            train_loader = torch.utils.data.DataLoader(train_data,
batch_size=self.batch_size, shuffle=True)
            epoch_loss = 0
            for train in train_loader:
                user_ratings = torch.Tensor(self.train[train[:,
0]]).to(self.device).to(torch.float32)
                item_is = train[:, 1].to(self.device) # pos_item_ids
                item_js = train[:, 2].to(self.device) # neg_item_ids

                '''
                Implement prediction and loss
```

```
            '''
            # IMPLEMENT HERE
            prediction_is = self.model.forward(user_ratings, item_is,
pos_item=True)
            prediction_js = self.model.forward(user_ratings, item_js,
pos_item=False)
            loss =  torch.pow((1 - (prediction_is - prediction_js)), 2).sum()
            #loss = torch.Tensor([0.0]).to(self.device)

            epoch_loss += loss.item() / len(train)

            # gradient reset
            self.optimizer.zero_grad()

            # Backpropagate
            loss.backward()

            # Update the parameters
            self.optimizer.step()

        if epoch % 1 == 0:
            top_k = 50
            prec, recall, ndcg, mrr, mAP = eval_implicit(self, self.train,
self.valid, top_k)
            print("[FISM] epoch %d, loss: %f" % (epoch,
epoch_loss/len(train_loader)))
            print(f"(FISM VALID) prec@{top_k} {prec}, recall@{top_k} {recall},
ndcg@{top_k} {ndcg}, mrr@{top_k} {mrr}, map@{top_k} {mAP}")
    # ========================= EDIT HERE ========================

def predict(self, user_id, item_ids):
    with torch.no_grad():
        user_rating =
torch.tensor([self.train[user_id]]).to(self.device).to(torch.float32)
        item_ids = torch.tensor(item_ids).to(self.device)
        prediction = self.model.forward(user_rating, item_ids, pos_item=False)
        return prediction.cpu().numpy()
```

(c) [20 pts] Write your code to implement the Embarassingly Shallow Autoencoders on 'models/EASE_implicit.py'. Please refer the algorithm in the paper (https://arxiv.org/pdf/1905.03375.pdf). You should not use gradient descent in here.

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

Answer:

```python
"""
Embarrassingly shallow autoencoders for sparse data,
Harald Steck,
Arxiv.
"""
import os
import math
import numpy as np

class EASE_implicit():
    def __init__(self, train, reg_lambda):
        self.train = train
        self.num_users = train.shape[0]
        self.num_items = train.shape[1]
        self.reg_lambda = reg_lambda

    def fit(self):
        '''
        Implement fit function
        '''
        self.B = np.zeros((self.num_users, self.num_items))
        # ========================= EDIT HERE ========================
        G = self.train.T.dot(self.train)
        diagIndices = np.diag_indices(G.shape[0])
        G[diagIndices] += self.reg_lambda
        P = np.linalg.inv(G)
        self.B = P / (-np.diag(P))
        self.B[diagIndices] = 0
        # ========================= EDIT HERE ========================
        # 사용자-항목 행렬과 W 행렬의 행렬 곱을 통해 예측 값 행렬 생성
        self.reconstructed = self.train @ self.B

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

(2) [20 pts] Using a reference code, fill out your evaluation metric code. Run '0_check.py' to validate your implementation code. Refer to MRR function in 'metrics.py', write your code to implement NDCG and MAP in the 'metrics.py'.

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

Answer:

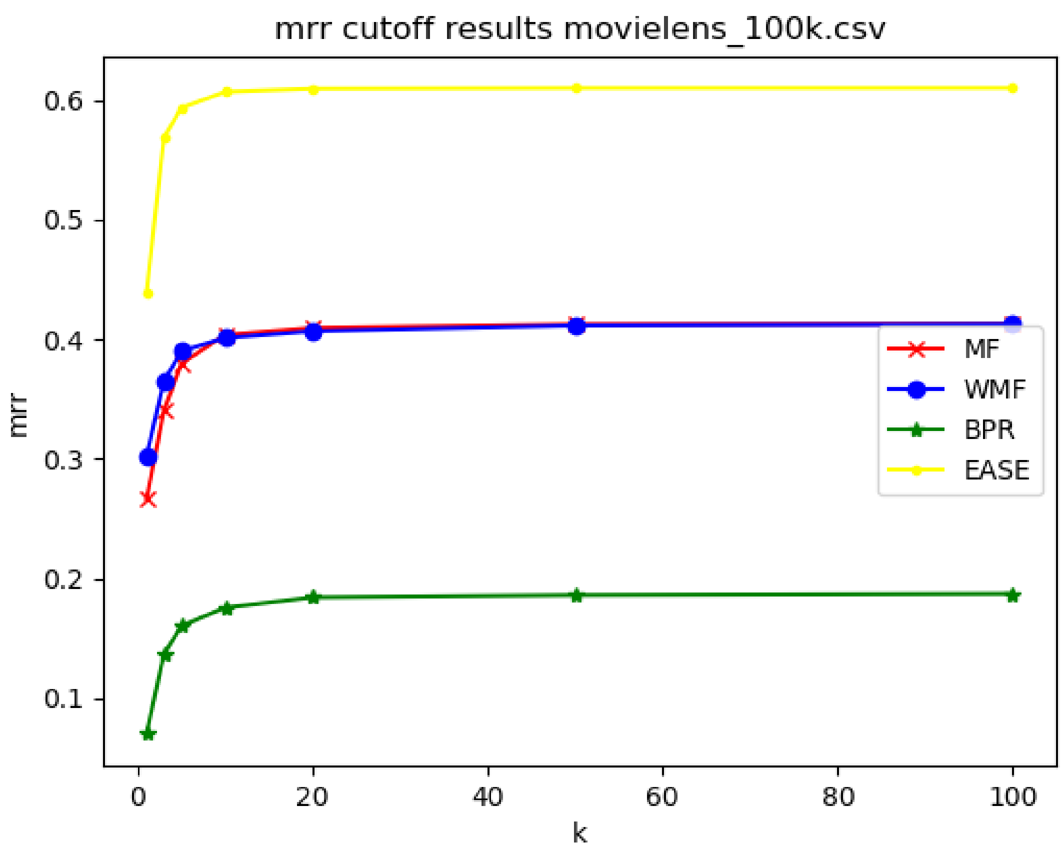
```
import math
import numpy as np


'''
# input
#    - pred_u: 예측 값으로 정렬 된 item index
#    - target_u: test set 의 item index
#    - top_k: top-k 에서의 k 값
'''
def compute_metrics(pred_u, target_u, top_k):
    pred_k = pred_u[:top_k] # 예측된 상위 k 개
    num_target_items = len(target_u)

    hits_k = [(i + 1, item) for i, item in enumerate(pred_k) if item in target_u]
    num_hits = len(hits_k)

    idcg_k = 0.0
    for i in range(1, min(num_target_items, top_k) + 1):
        idcg_k += 1 / math.log(i + 1, 2)

    dcg_k = 0.0
    for idx, item in hits_k:
        dcg_k += 1 / math.log(idx + 1, 2)

    prec_k = num_hits / top_k
    recall_k = num_hits / min(num_target_items, top_k)
    ndcg_k = dcg_k / idcg_k

    '''
    Implement RR@K and AP@K in here
    '''
    # ========================= EDIT HERE ========================
    if len(hits_k) == 0: # 맞춘게 없을 경우
        rr_k = 0
    else:
        rr_k =  1 / hits_k[0][0]

    if len(hits_k) == 0: # 맞춘게 없을 경우
        ap_k = 0
    else:
```

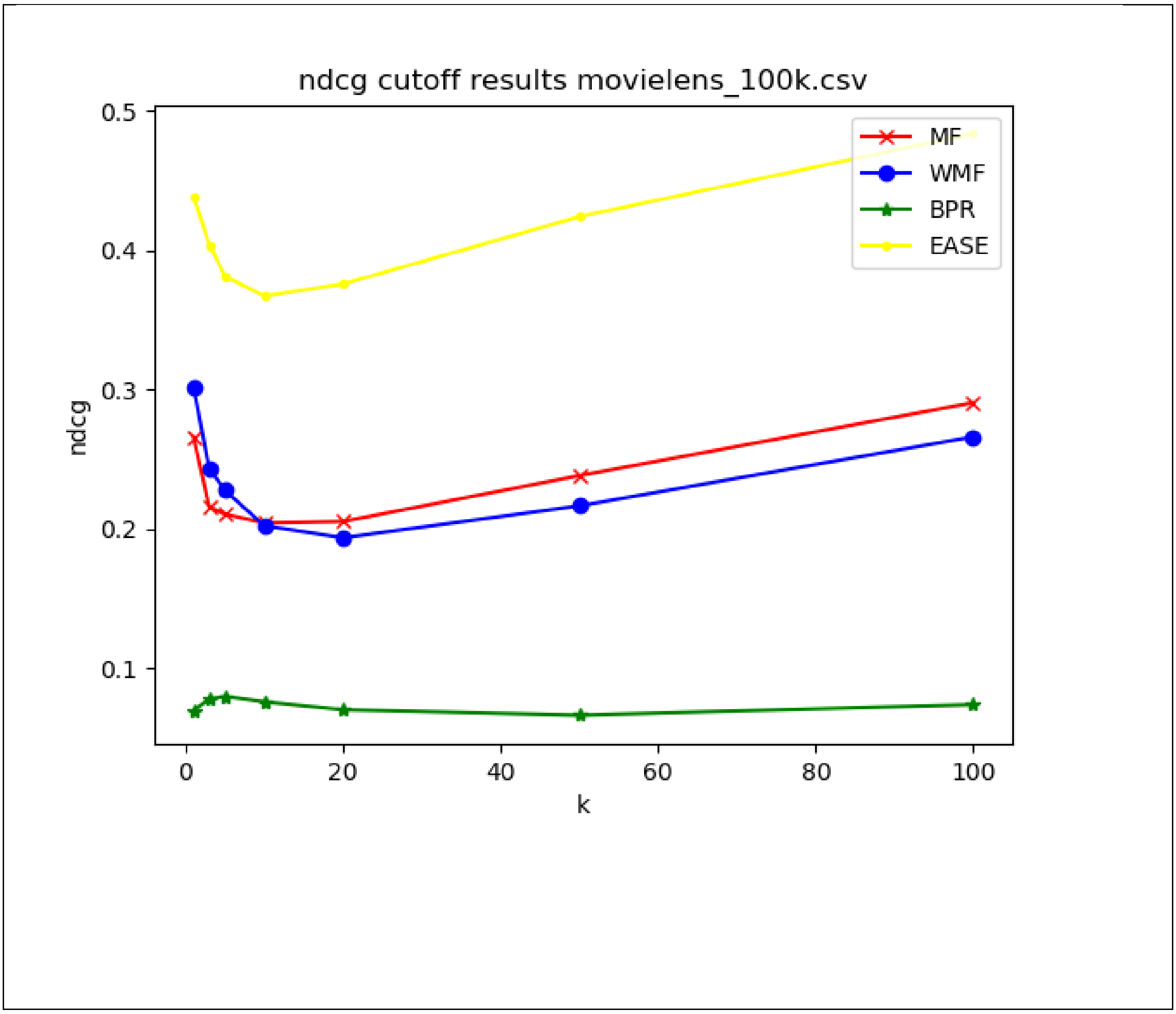
```python
        precs = 0
        for i, hit in enumerate(hits_k):
            prec = (i + 1) / hit[0]
            precs += prec
        ap_k = precs / num_hits
    # ==============================================================
    return prec_k, recall_k, ndcg_k, rr_k, ap_k

'''
You can implement metrics using follow functions if you want
'''
def get_rr_k(pred_u, target_u, top_k):
    # ========================= EDIT HERE ========================
    rr_k = -1
    # ==============================================================
    return rr_k

def get_ap_k(pred_u, target_u, top_k):
    # ========================= EDIT HERE ========================
    ap_k = -1
    # ==============================================================
    return ap_k
```

(3) [20 pts] Given the data ('naver_movie_dataset_100k.csv' and 'movielens_1m.csv'), draw the plots of MRR, NDCG, and MAP by adjusting cutoff at MF, BPR, FISM, and EASE. With adjusting cutoff sizes (number of recommended items), explain the results . Run '2_search.py' to run the code.

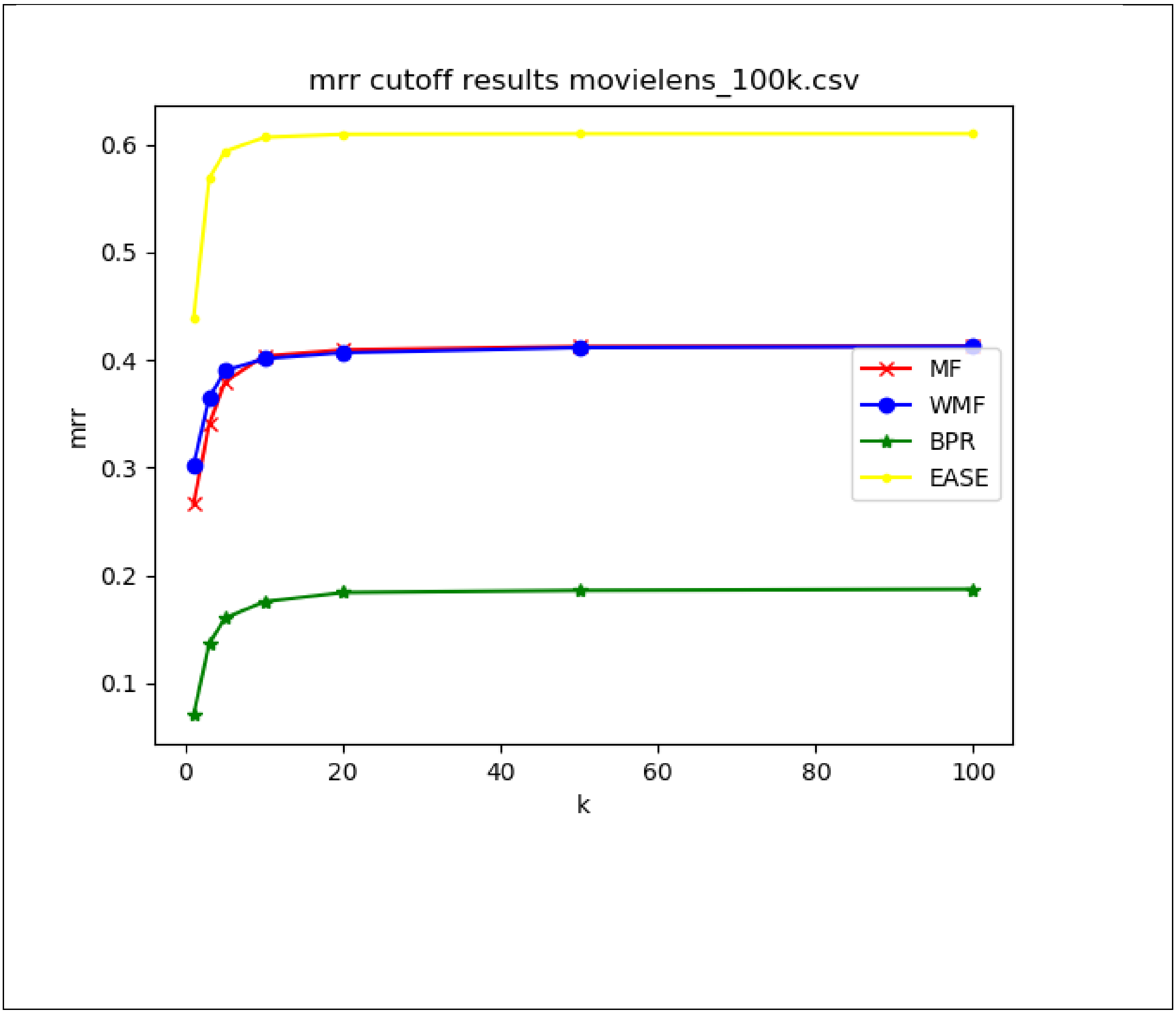
Note: Please show your plots for two datasets and explanations in short (3-5) lines.
(e.g., A model shows the best performance over all metrics for the lower cutoff, but B model shows the best NDCG when the cutoff is high.)

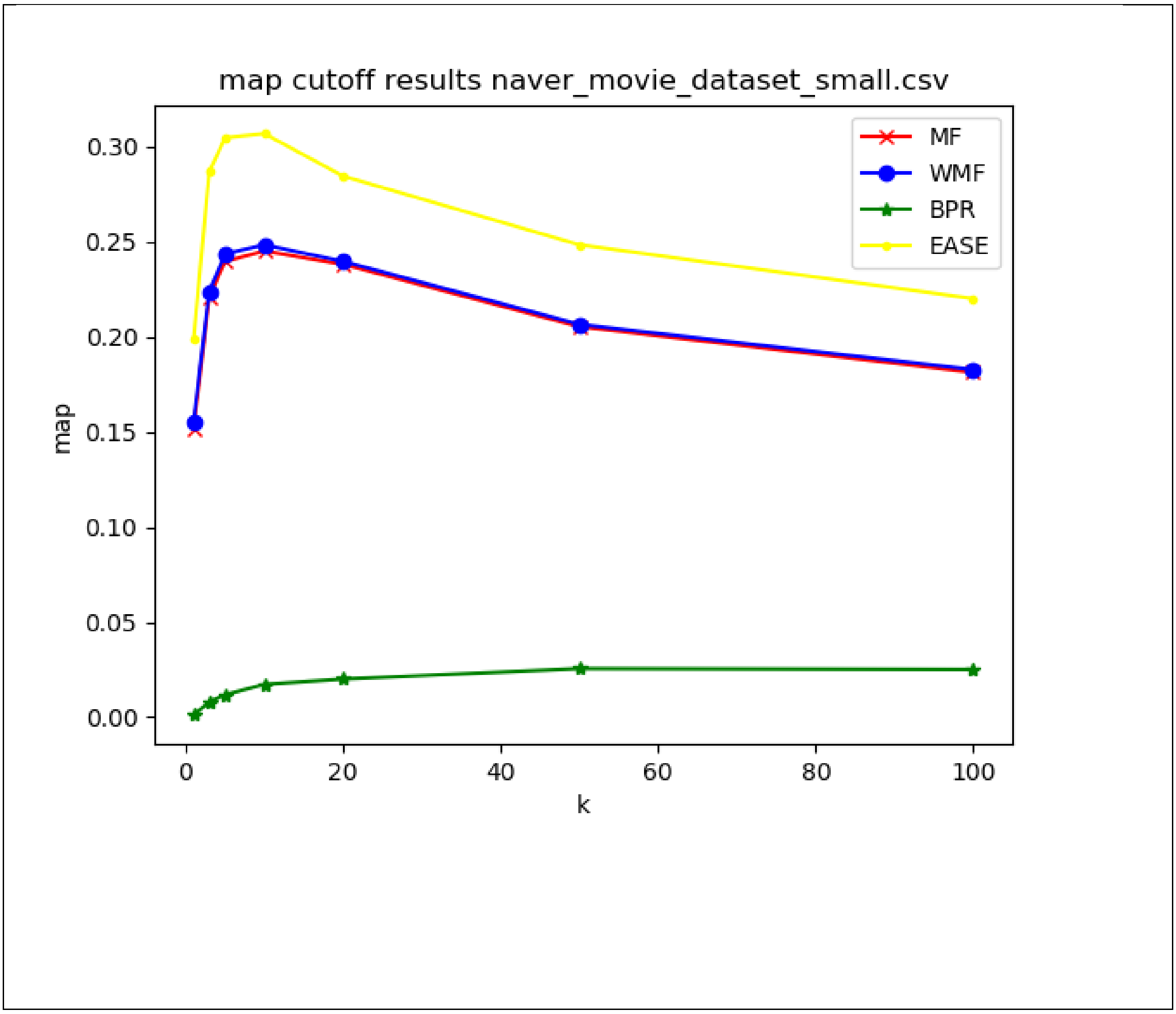
Answer:

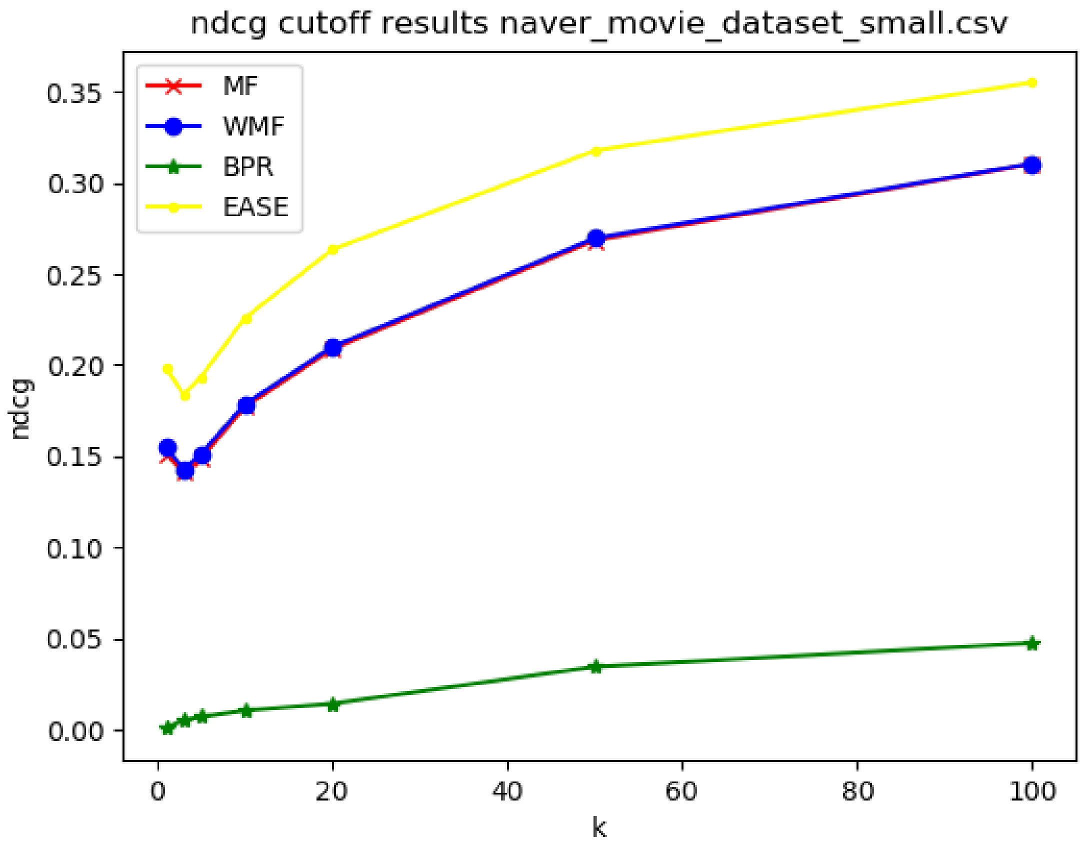




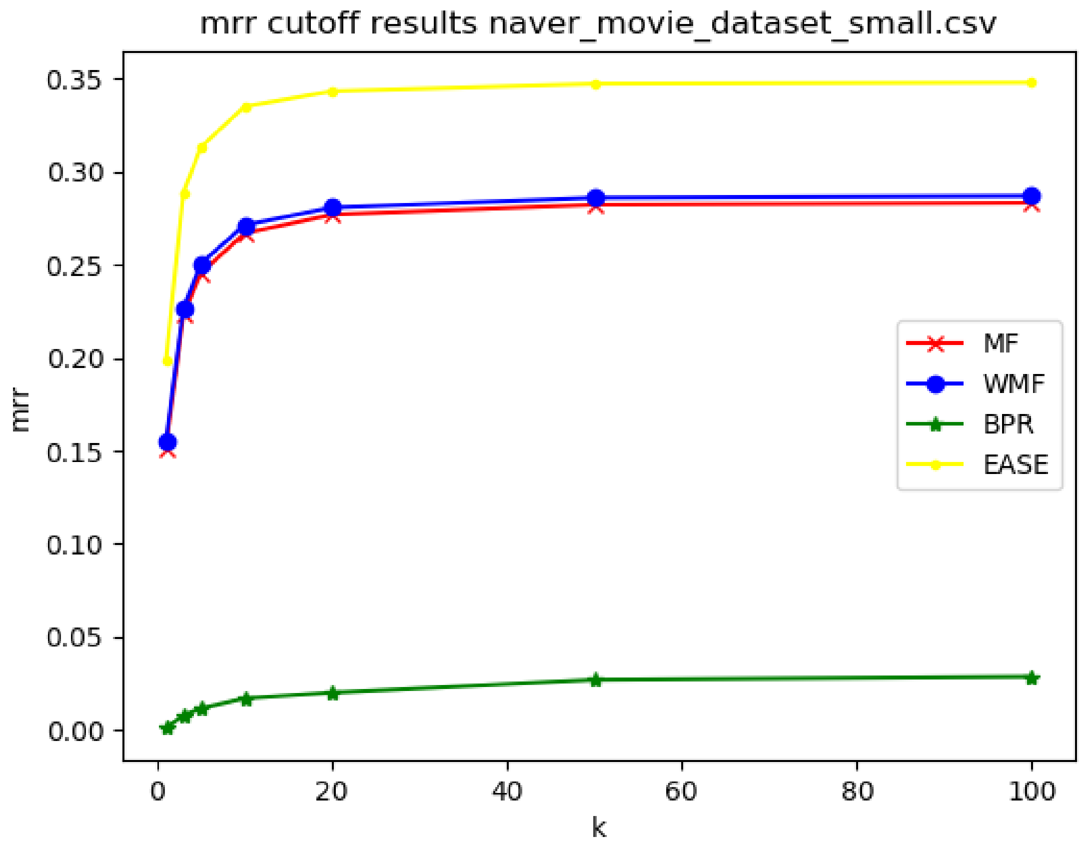
ndcg cutoff results movielens_100k.csv
ndcg
k
MF
WMF
BPR
EASE
0.5
0.4
0.3
0.2
0.1
0
20
40
60
80
100


mrr cutoff results movielens_100k.csv
mrr
k
0.1
0.2
0.3
0.4
0.5
0.6
0
20
40
60
80
100
MF
WMF
BPR
EASE


map cutoff results naver_movie_dataset_small.csv
MF
WMF
BPR
EASE
map
k
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
20
40
60
80
100



ndcg cutoff results naver_movie_dataset_small.csv
MF
WMF
BPR
EASE
ndcg
k
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0
20
40
60
80
100



MF, WMF, BPR, and EASE show better performance when cutoff is from 10 to 20. And MF, WMF,BPR and EASE show few change when cutoff increase from 20.