final_project

December 12, 2022

1 preprocessing

In this homework assignment, you will extend your hw01 and hw02. Your goal is to incorporate two different graph-based neural network methods to boost the recommendation performance. You will test your new implementation using the same Netflix Prize data.

We have covered several graph-based neural network methods in our class and readings, including shallow and deep approaches, knowledge graph appraoches, etc. Pick two distinctive methods that you are interested in, implement and evaluate their performance on the Netflix movie recommendation dataset. In your final report, explain why you select the methods, and provide a rationale for why you think the method may potentially improve your recommender system.

Use the best two models from your previous assignments (hw01, hw02) as your baseline.

(B1) your best model from hw01-02; describe what the model is (B2) your 2nd best model from hw01-02; describe what the model is (A1) your first graph-based neural network model (A2) your first graph-based neural network model Compare the model performance based on both implicit and explicit feedback. This allows you to understand the strength of your models. For explicit feedback, use MAPE and RMSE as evaluation metrics (as in hw01), and for implicit feedback, use HR and NDCG (as in hw02). In your final report, provide a detailed performance analysis and discuss the strengths and weaknesses of your models.

Note: If you use anyone's shared code or build your model based on any existing repositories, make sure you include all of them in your acknowledgment and references.

Your submission will include (1) a reproducible notebook, (2) project presentation slides, and (3) a final paper. See final project guideline for more details.

[ ]:
```
from datetime import datetime
# globalstart = datetime.now()
import pandas as pd
import numpy as np
import matplotlib
matplotlib.use('nbagg')
import random
import matplotlib.pyplot as plt
plt.rcParams.update({'figure.max_open_warning': 0})
import seaborn as sns
sns.set_style('whitegrid')
import os
```

```
import random
import math
import pickle
import time

import torch
!pip install torchmetrics
import torchmetrics
from torch import nn
from torch import optim
from torch.utils.data import DataLoader
from torchvision import datasets
from torch.nn.functional import mse_loss
from torchmetrics import MeanAbsolutePercentageError
from gensim.models import Word2Vec
import multiprocessing
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Collecting torchmetrics
  Downloading torchmetrics-0.11.0-py3-none-any.whl (512 kB)
     |                  | 512 kB 4.1 MB/s
Requirement already satisfied: numpy>=1.17.2 in
/usr/local/lib/python3.8/dist-packages (from torchmetrics) (1.21.6)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.8/dist-packages (from torchmetrics) (4.4.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.8/dist-
packages (from torchmetrics) (21.3)
Requirement already satisfied: torch>=1.8.1 in /usr/local/lib/python3.8/dist-
packages (from torchmetrics) (1.13.0+cu116)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in
/usr/local/lib/python3.8/dist-packages (from packaging->torchmetrics) (3.0.9)
Installing collected packages: torchmetrics
Successfully installed torchmetrics-0.11.0
```

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/MyDrive/final_netflix_project
!ls
```

```
Mounted at /content/drive
/content/drive/MyDrive/final_netflix_project
deepwalk_weight.model  G_unweighted.pickle  models           node2vec.model
final_project.ipynb    G_weighted.pickle    Netflix_dataset
```

[ ]:
```
#Preprocessing
##Converting / Merging whole data to required format: u_i, m_j, r_ij
```

```
start = datetime.now()
if not os.path.isfile('Netflix_dataset/data.csv'):
    # Creating a file 'data.csv' before reading it
    # Read all the files in netflix and store them in one big file('data.csv')
    # We re reading from each of the four files and appendig each rating to a␣
 ↪global file 'train.csv'
    data = open('Netflix_dataset/data.csv', mode='w')

    row = list()
    files = ['Netflix_dataset/combined_data_1.txt', 'Netflix_dataset/
 ↪combined_data_2.txt',
             'Netflix_dataset/combined_data_3.txt', 'Netflix_dataset/
 ↪combined_data_4.txt']

    for file in files:
        print(1)
        print("Reading ratings from {}...".format(file))

        with open(file) as f:
            for line in f:
                del row[:]  # We might not have to do this.
                line = line.strip()
                if line.endswith(':'):
                    # All below are ratings for this movie, until another movie␣
 ↪appears.
                    movie_id = line.replace(':', '')
                else:
                    row = [x for x in line.split(',')]
                    row.insert(0, movie_id)
                    data.write(','.join(row))
                    data.write('\n')
        print("Done.\n")

    data.close()

print('Time taken :', datetime.now() - start)

print("creating the dataframe from data.csv file..")
df = pd.read_csv('Netflix_dataset/data.csv', sep=',',names=['movie',␣
 ↪'user','rating','date'])
df.date = pd.to_datetime(df.date)
print('Done.\n')
# we are arranging the ratings according to time.
print('Sorting the dataframe by date..')
df.sort_values(by='date', inplace=True)
print('Done..')
```

```
print(df.head())

print(df.describe()['rating'])
```

```
Time taken : 0:00:00.544095
creating the dataframe from data.csv file..
Done.

Sorting the dataframe by date..
Done..
           movie    user  rating       date
56431994  10341  510180       4 1999-11-11
9056171    1798  510180       5 1999-11-11
58698779  10774  510180       3 1999-11-11
48101611   8651  510180       2 1999-11-11
81893208  14660  510180       2 1999-11-11
count    1.004805e+08
mean     3.604290e+00
std      1.085219e+00
min      1.000000e+00
25%      3.000000e+00
50%      4.000000e+00
75%      4.000000e+00
max      5.000000e+00
Name: rating, dtype: float64
```

[ ]:
```
#checking NaN values
# just to make sure that all Nan containing rows are deleted..
print("No of Nan values in our dataframe : ", sum(df.isnull().any()))

#Removing Duplicates
dup_bool = df.duplicated(['movie','user','rating'])
dups = sum(dup_bool) # by considering all columns..( including timestamp)
print("There are {} duplicate rating entries in the data..".format(dups))
```

```
No of Nan values in our dataframe :  0
There are 0 duplicate rating entries in the data..
```

[ ]:
```
#choose subdataset based on the id of user
new_df = df[df['user']<df['user'].max()/20]
new_df.to_csv('Netflix_dataset/new_data.csv',index=False)
```

[ ]:
```
new_df = pd.read_csv('Netflix_dataset/new_data.csv', sep=',',names=['movie',␣
↪'user','rating','date'])
```

```
/usr/local/lib/python3.8/dist-packages/IPython/core/interactiveshell.py:3326:
DtypeWarning: Columns (0,1,2) have mixed types.Specify dtype option on import or
set low_memory=False.
```

```
  exec(code_obj, self.user_global_ns, self.user_ns)
```

[ ]:
```
new_df.describe()
```

[ ]:
```
           movie     user   rating        date
count    4985661  4985661  4985661     4985661
unique     22176    24960       11        2166
top         5317    16272        4  2005-01-19
freq       11499     5900  1635230       37005
```

[ ]:
```
#Spliting data into Train and Test(80:20)
if not os.path.isfile('Netflix_dataset/train.csv'):
    # create the dataframe and store it in the disk for offline purposes..
    new_df.iloc[:int(new_df.shape[0]*0.80)].to_csv("Netflix_dataset/train.csv",␣
 ↪index=False)

if not os.path.isfile('Netflix_dataset/test.csv'):
    # create the dataframe and store it in the disk for offline purposes..
    new_df.iloc[int(new_df.shape[0]*0.80):].to_csv("Netflix_dataset/test.csv",␣
 ↪index=False)

train_df = pd.read_csv("Netflix_dataset/train.csv", parse_dates=['date'])
test_df = pd.read_csv("Netflix_dataset/test.csv", parse_dates=['date'])
```

```
/usr/local/lib/python3.8/dist-packages/IPython/core/interactiveshell.py:3326:
DtypeWarning: Columns (0,1,2) have mixed types.Specify dtype option on import or
set low_memory=False.
  exec(code_obj, self.user_global_ns, self.user_ns)
```

[ ]:
```
print('length of train_df:',len(train_df))
print('length of test_df',len(test_df))
print('length of new_df',len(new_df))
```

```
length of train_df: 3988528
length of test_df 997133
length of new_df 4985661
```

[ ]:
```
import pandas as pd
from collections import defaultdict
import os
import csv
import sys
import math
import numpy as np
```

1.1 transfer data to be stored in dictionary

[ ]:
```
def create_user_movie_dictionary(train_df,test_df,train_path,test_path):
    if not os.path.isfile(train_path):
        user_movie_train = defaultdict(list)
        for iter, row in train_df.iterrows():
            #user-movie: key-user; value-[movie,rating]
            user_movie_train[row[1]].append([row[0], row[2]])
        with open(train_path, 'w', encoding='utf-8-sig') as f:
            w = csv.writer(f)
            header = ['user', 'movie_and_rating']
            w.writerow(header)
            w.writerows(user_movie_train.items())
    else:
        user_movie_train = pd.read_csv(train_path).
 ↪set_index('user')['movie_and_rating'].to_dict()
        del user_movie_train['user']
        for user in user_movie_train.keys():
          user_movie_train[user]=eval(user_movie_train[user])
    if not os.path.isfile(test_path):
        user_movie_test = defaultdict(list)
        for iter, row in test_df.iterrows():
            #user-movie: key-user; value-[movie,rating]
            user_movie_test[row[1]].append([row[0], row[2]])
        with open(test_path, 'w', encoding='utf-8-sig') as f:
            w = csv.writer(f)
            header = ['user', 'movie_and_rating']
            w.writerow(header)
            w.writerows(user_movie_test.items())
    else:
        user_movie_test = pd.read_csv(test_path).
 ↪set_index('user')['movie_and_rating'].to_dict()

        for user in user_movie_test.keys():
          user_movie_test[user]=eval(user_movie_test[user])
    return user_movie_train,user_movie_test
user_movie_train,user_movie_test=create_user_movie_dictionary(train_df,test_df,"Netflix_datase
 ↪user_movie_dictionary_train.csv","Netflix_dataset/user_movie_dictionary_test.
 ↪csv")
```

[ ]:
```
def create_movie_user_dictionary(train_df,test_df,train_path,test_path):
    if not os.path.isfile(train_path):
        movie_user_train = defaultdict(list)
        for iter, row in train_df.iterrows():
            #movie-user: key-movie; value-[movie,rating]
            movie_user_train[row[0]].append([row[1], row[2]])
        with open(train_path, 'w', encoding='utf-8-sig') as f:
```

```
            w = csv.writer(f)
            header = ['movie', 'user_and_rating']
            w.writerow(header)
            w.writerows(movie_user_train.items())
    else:
        movie_user_train = pd.read_csv(train_path).
↪set_index('movie')['user_and_rating'].to_dict()
        del movie_user_train['movie']
        for movie in movie_user_train.keys():
          movie_user_train[movie]=eval(movie_user_train[movie])
    if not os.path.isfile(test_path):
        movie_user_test = defaultdict(list)
        for iter, row in test_df.iterrows():
            #user-movie: key-user; value-[movie,rating]
            movie_user_test[row[0]].append([row[1], row[2]])
        with open(test_path, 'w', encoding='utf-8-sig') as f:
            w = csv.writer(f)
            header = ['movie', 'user_and_rating']
            w.writerow(header)
            w.writerows(movie_user_test.items())
    else:
        movie_user_test = pd.read_csv(test_path).
↪set_index('movie')['user_and_rating'].to_dict()
        for movie in movie_user_test.keys():
          movie_user_test[movie]=eval(movie_user_test[movie])
    return movie_user_train,movie_user_test
movie_user_train,movie_user_test=create_movie_user_dictionary(train_df,test_df,"Netflix_datase
↪movie_user_dictionary_train.csv","Netflix_dataset/movie_user_dictionary_test.
↪csv")
```

2 transform data to implicit data

[ ]:
```
"""
for user_movie_test dictionary, add 100 negative samples to each user,
choose negative samples: movie in all_movies of new_df, but not in this user's␣
↪watching history
add to the user's watching history


"""
```

[ ]:
```
"\nfor user_movie_test dictionary, add 100 negative samples to each
user,\nchoose negative samples: movie in all_movies of new_df, but not in this
user's watching history\nadd to the user's watching history\n\n\n"
```

[ ]:
```
output_path = 'Netflix_dataset/test_df_implicit.csv'
if not os.path.isfile(output_path):
    test_users = test_df['user'].unique()
    test_df_implicit_dict = {'user':[],'movie':[],'rating':[]}
    new_df = new_df.iloc[1:]
    all_movies = set(new_df['movie'].unique())
    print(len(all_movies))
    for user in user_movie_test.keys():

        test_df_implicit_dict['user'].append(user)
        test_df_implicit_dict['movie'].append(user_movie_test[user][-1][0])
        test_df_implicit_dict['rating'].append(1)
        positive_movies = [item[0] for item in user_movie_test[user]]
        negative_movies = random.sample(list(all_movies.
 ↪difference(set(positive_movies))),100)
        for movie in negative_movies:
            test_df_implicit_dict['user'].append(user)
            test_df_implicit_dict['movie'].append(movie)
            test_df_implicit_dict['rating'].append(0)
    test_df_implicit= pd.DataFrame(test_df_implicit_dict)
    test_df_implicit.to_csv(output_path,index=False)
else:
    test_df_implicit= pd.read_csv(output_path)
#test_df_implicit = test_df_implicit.iloc[1:]
#test_df_implicit=test_df_implicit.
 ↪drop(test_df_implicit[test_df_implicit['movie']=='movie'].index)
test_df_implicit.head()
```

[ ]:

	user	movie	rating
0	40563	1435	1
1	40563	13303	0
2	40563	14372	0
3	40563	14536	0
4	40563	12950	0

[ ]:
```
output_path = 'Netflix_dataset/train_df_implicit.csv'
if not os.path.isfile(output_path):
    train_users = train_df['user'].unique()
    train_df_implicit_dict = {'user':[],'movie':[],'rating':[]}
    new_df = new_df.iloc[1:]
    all_movies = set(new_df['movie'].unique())
    print(len(all_movies))
    for user in user_movie_train.keys():

        train_df_implicit_dict['user'].append(user)
        train_df_implicit_dict['movie'].append(user_movie_train[user][-1][0])
        train_df_implicit_dict['rating'].append(1)
```

```
        positive_movies = [item[0] for item in user_movie_train[user]]
        negative_movies = random.sample(list(all_movies.
 ↪difference(set(positive_movies))),10)
        for movie in negative_movies:
            train_df_implicit_dict['user'].append(user)
            train_df_implicit_dict['movie'].append(movie)
            train_df_implicit_dict['rating'].append(0)
    train_df_implicit= pd.DataFrame(train_df_implicit_dict)
    train_df_implicit.to_csv(output_path,index=False)
else:
    train_df_implicit= pd.read_csv(output_path)
#test_df_implicit = test_df_implicit.iloc[1:]
#test_df_implicit=test_df_implicit.
 ↪drop(test_df_implicit[test_df_implicit['movie']=='movie'].index)
train_df_implicit.head()
```

[ ]:

	user	movie	rating
0	122223	7379	1
1	122223	3842	0
2	122223	5003	0
3	122223	4909	0
4	122223	5236	0

3 create network

[ ]:
```
users = set(new_df['user'].unique())
movies = set(new_df['movie'].unique())
len(users & movies)
```

[ ]: 3213

There's duplication between users and movies, so change the users'name to be U..., and change the movies's name to be M...

[ ]:
```
#add nodes to the network
nodes = []
for user in users:
  nodes.append('U'+str(user))
for movie in movies:
  nodes.append('M'+str(movie))
```

[ ]:
```
#add edges to the network(only for train dataset)
edges_unweighted = []
edges_weighted = []
for user in user_movie_train.keys():
```

```
  movie_and_rating = user_movie_train[user]
  for movie, rating in movie_and_rating:
    edges_unweighted.append(['U'+str(user),'M'+str(movie)])
    edges_unweighted.append(['M'+str(movie),'U'+str(user)])
    edges_weighted.append(['U'+str(user),'M'+str(movie),rating])
    edges_weighted.append(['M'+str(movie),'U'+str(user),rating])
```

[ ]:
```
import networkx as nx
G_weighted = nx.Graph()
G_weighted.add_nodes_from(nodes)
G_weighted.add_weighted_edges_from(edges_weighted)
```

[ ]:
```
G_unweighted = nx.Graph()
G_unweighted.add_nodes_from(nodes)
G_unweighted.add_edges_from(edges_unweighted)
```

[ ]:
```
import pickle
pickle.dump(G_weighted, open('G_weighted.pickle', 'wb'))
pickle.dump(G_unweighted, open('G_unweighted.pickle','wb'))
```

[ ]:
```
# load graph object from file
G_unweighted = pickle.load(open('G_unweighted.pickle', 'rb'))
G_weighted = pickle.load(open('G_weighted.pickle','rb'))
```

[ ]:
```
# """
# for G_weighted:add the edge between users and users based on the similarity␣
 ↪betwee users
# (1)movie_of_user1 & movie_of_user2 = movie_both
# (2)for movie_both: calculate total value of each movie in user1 and user2
# (3)calculate the total value of all movies of user1 and user2
# (4)use (2)/(3)
# """
# def calculate_similarity_between_users(user_movie_train):
#   edges = []
#   count = 0
#   for user1 in user_movie_train:
#     count += 1
#     if count%1000 == 0:
#       print(count)
#     for user2 in user_movie_train:
#       if user1 == user2: continue
#       movies_of_user1 = [movie_rating[0] for movie_rating in␣
 ↪user_movie_train[user1]]
#       movies_of_user2 = [movie_rating[0] for movie_rating in␣
 ↪user_movie_train[user2]]
#       ratings_of_user1 = [movie_rating[1] for movie_rating in␣
 ↪user_movie_train[user1]]
```

```
#       ratings_of_user2 = [movie_rating[1] for movie_rating in␣
 ↪user_movie_train[user2]]
#       movies_of_both = list(set(movies_of_user1)&set(movies_of_user2))
#       score_of_both = 0
#       score_of_total = 0
#       for i in range(len(movies_of_user1)):
#         movie, rating = movies_of_user1[i],ratings_of_user1[i]
#         if movie in movies_of_both:
#           score_of_both += float(rating)
#         score_of_total += float(rating)
#       for i in range(len(movies_of_user2)):
#         movie, rating = movies_of_user2[i],movies_of_user2[i]
#         if movie in movies_of_both:
#           score_of_both += float(rating)
#         score_of_total += float(rating)
#       similarity = score_of_both / score_of_total
#       if similarity >= 0.25:
#         edges.append(['U'+str(user1),'U'+str(user2)])
#         edges.append(['U'+str(user2),'U'+str(user1)])
#   return edges
# new_edges = calculate_similarity_between_users(user_movie_train)
```

[ ]:

[ ]:
```
# """
# add the edge between users and users based on the similarity betwee users
# (1)movie_of_user1 & movie_of_user2
# (2)movie_of_user1 || movie_of_user2
# (3) (1)/(2) > 0.25:similarity (ab,ac a/abc)
# """
# def calculate_similarity_between_users_implicit(user_movie_train_implicit):
#   edges = []
#   for user1 in user_movie_train:
#     for user2 in user_movie_train:
#       if user1 == user2: continue
#       movies_of_user1 = [movie_rating[0] for movie_rating in␣
 ↪user_movie_train[user1]]
#       movies_of_user2 = [movie_rating[0] for movie_rating in␣
 ↪user_movie_train[user2]]
#       movies_of_both = list(set(movies_of_user1)&set(movies_of_user2))
#       movies_of_each = list(set(movies_of_user1)|set(movies_of_user2))
#       similarity = len(movies_of_both)/len(movies_of_each)
#       if similarity >= 0.1:
#         edges.append(['U'+str(user1),'U'+str(user2)])
#         edges.append(['U'+str(user2),'U'+str(user1)])

#   return edges
```

```
# new_edges = calculate_similarity_between_users_implicit(user_movie_train_implicit)
```

[ ]:

4 deep walk

[ ]:
```
from gensim.models import Word2Vec
import multiprocessing
```

[ ]:
```
class DeepWalk:
  def __init__(self, G):
    self.G = G
    self.nodes = list(G.nodes())
    self.representation = {}
    k = 32 #representation dimention
    w = 5 #window size
    step = 1 #run how many times for each node
    learning_rate = 0.05
    walk_length = 80
    self.workers = multiprocessing.cpu_count()
    self.model = self.DeepWalk(w, k, step, walk_length, learning_rate)
    self.get_representation()
  def RandomWalk(self,start_node,walk_length):
    walk = [start_node]
    while len(walk) < walk_length:
          cur = walk[-1]
          cur_nbrs = list(self.G.neighbors(cur))
          if len(cur_nbrs) > 0:
              walk.append(random.choice(cur_nbrs))
          else:
              break
    return walk

  def learn_representation(self,walk,k,w,learning_rate):#k:dimention of representation; w: window size
    # for node in walk:
    #   self.representation[node].setdefault(node, np.random.random((k,1))/10*np.sqrt(k))
    #eg. walk = [1,2,3,4,5], w = 2 new_walk = [4,5]+[1,2,3,4,5]+[1,2]
    walk = walk[-w:] + walk + walk[:w]
    for i in range(w,len(walk)-w): #2,7
      for j in range(i-w,i+w+1):#0,1, ,3,4
        if j == i: continue
        diff = self.representation[walk[i]] - self.representation[walk[j]]
```

```
        self.representation[walk[j]] += learning_rate * diff/self.
↪representation[walk[j]]
 def get_representation(self):
   for node in self.nodes:
     self.representation[node] = self.model.wv.get_vector(node)
 def DeepWalk(self, w, k, step, walk_length, learning_rate):
   # for node in self.G.nodes:
   #   self.representation.setdefault(node, np.random.random((k,1)))
   walks = []
   random.shuffle(list(self.G.nodes))
   count = 0
   for start_node in self.G.nodes:
     count += 1
     if count % 10000 == 0:
       print(count)
     walk = self.RandomWalk(start_node, walk_length)
     walks.append(walk)
   #self.learn_representation(walk, k, w, learning_rate)
   self.model = Word2Vec(walks,window=w,min_count=1,workers=self.
↪workers,size=k,iter=step,sg=1)
   return self.model

deepwalk = DeepWalk(G_unweighted)
```

```
10000
20000
30000
40000
```

[ ]:
```
!pip install torchmetrics
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: torchmetrics in /usr/local/lib/python3.8/dist-
packages (0.11.0)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.8/dist-packages (from torchmetrics) (4.4.0)
Requirement already satisfied: numpy>=1.17.2 in /usr/local/lib/python3.8/dist-
packages (from torchmetrics) (1.21.6)
Requirement already satisfied: packaging in /usr/local/lib/python3.8/dist-
packages (from torchmetrics) (21.3)
Requirement already satisfied: torch>=1.8.1 in /usr/local/lib/python3.8/dist-
packages (from torchmetrics) (1.13.0+cu116)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in
/usr/local/lib/python3.8/dist-packages (from packaging->torchmetrics) (3.0.9)
```

[ ]:
```
import torch
import torchmetrics
from torch import nn
from torch import optim
from torch.utils.data import DataLoader
from torchvision import datasets
from torch.nn.functional import mse_loss
from torchmetrics import MeanAbsolutePercentageError
```

[ ]:
```

```

4.0.1 explicit

[ ]:
```
from torch.utils.data import Dataset

class PandasDataset(Dataset):
    def __init__(self, dataframe):
        self.dataframe = dataframe
        self.user_list = list(dataframe['user'])
        self.movie_list = list(dataframe['movie'])
        self.rating_list = list(dataframe['rating'])
        if 'user' in self.user_list:
          self.user_list.remove('user')
        if 'movie' in self.movie_list:
          self.movie_list.remove('movie')
        if 'rating' in self.rating_list:
          self.rating_list.remove('rating')
    def __len__(self):
        return len(self.dataframe)

    def __getitem__(self, idx):
      user = int(self.user_list[idx])
      movie = int(self.movie_list[idx])
      rating = float(self.rating_list[idx])
      return (
      torch.tensor(user, dtype=torch.long),
      torch.tensor(movie, dtype=torch.long),
      torch.tensor(rating, dtype=torch.float)
      )


train_df = train_df.drop(train_df[train_df['user']=='user'].index)
test_df = test_df.drop(test_df[test_df['user']=='user'].index)
train_dataset = PandasDataset(train_df)
test_dataset = PandasDataset(test_df)
```

[ ]:
```
LR = 0.001
EPOCHS = 20
BATCH_SIZE = 100
LR_DECAY_STEP = 5
LR_DECAY_VALUE = 10
```

[ ]:
```
class Model(torch.nn.Module):
  def __init__(self, deepwalk,users,movies,k,batch_size):
    super(Model, self).__init__()
    self.G = deepwalk.G
    self.representation = deepwalk.representation
    for node in self.G.nodes:
      self.representation.setdefault(node, np.random.random((k,1))/10*np.
↪sqrt(k))
    self.k = k
    self.users = users
    self.movies = movies
    self.batch_size = batch_size

    input_size = k*2
    hidden_size1 = [128, 48]
    output_size = 1
    #self.layer1 = torch.nn.Linear(input_size, hidden_size1[0]),
    #self.layer2 = nn.Linear(hidden_size1[0], hidden_size1[1]),
    #self.layer3 = nn.Linear(hidden_size1[1],output_size),
    self.model = nn.ModuleList()
    self.model.append(torch.nn.Linear(input_size, hidden_size1[0]))
    #self.model.append(torch.nn.ReLU())
    self.model.append(torch.nn.Linear(hidden_size1[0], hidden_size1[1]))
    #self.model.append(torch.nn.ReLU())
    self.model.append(torch.nn.Linear(hidden_size1[1],output_size))
    #self.model.append(torch.nn.Softmax())
    print(self.model)
    self.reset_parameters()
  def reset_parameters(self):
      for i in self.model:
        if isinstance(i,nn.ReLU) or isinstance(i,nn.Softmax):continue
        i.reset_parameters()
  def predict(self,user, movie):
    temp = np.concatenate((self.representation[f'U{user}'],self.
↪representation[f'M{movie}']),axis=0)
    y_pred = self.model(torch.tensor(np.transpose(temp)).float())
    #print(y_pred)
    return y_pred
  def forward(self, user_indices, movie_indices):
    #user_embedding_mlp = self.embedding_user_mlp(user_indices)
    #item_embedding_mlp = self.embedding_item_mlp(item_indices)
```

```
    user,movie = user_indices[0], movie_indices[0]
    user_embedding = torch.tensor(self.representation[f'U{user}'])
    movie_embedding = torch.tensor(self.representation[f'M{movie}'])
    embedding_of_batch = torch.cat([user_embedding, movie_embedding], dim=-1).
↪reshape((2*self.k,1))
    for i in range(1, len(user_indices)):
        user,movie = user_indices[i], movie_indices[i]
        user_embedding = torch.tensor(self.representation[f'U{user}']).float()
        movie_embedding = torch.tensor(self.representation[f'M{movie}']).float()
        temp = torch.cat([user_embedding, movie_embedding], dim=-1).
↪reshape((2*self.k,1))
        embedding_of_batch = torch.cat([embedding_of_batch, temp], dim=-1)
    embedding_of_batch = torch.t(embedding_of_batch)
    #print(embedding_of_batch.shape)
    for idx, _ in enumerate(range(len(self.model))):
        #print(idx, embedding_of_batch)
        embedding_of_batch = self.model[idx](embedding_of_batch)
        #print(embedding_of_batch.shape)
    #print(embedding_of_batch)
    result = embedding_of_batch.squeeze()
    return result
```

[ ]:
```
train_loader = DataLoader(train_dataset, BATCH_SIZE, shuffle=True,␣
↪num_workers=2)
test_loader = DataLoader(test_dataset, BATCH_SIZE, shuffle=False, num_workers=2)
```

[ ]:
```
users_train = list(user_movie_train.keys())
movies_train = list(movie_user_train.keys())
model = Model(deepwalk,users_train,movies_train,k=32,batch_size=100)
```

[ ]:
```
#model.to(device)
model.reset_parameters()
opt = optim.Adam(model.parameters(), lr=LR)
```

[ ]:
```
EPOCHS = 20
start_time = time.time()
for epoch in range(1, EPOCHS+1):
    model.train()
    train_loss_all = 0
    for user, item, label in train_loader:
      opt.zero_grad()
      prediction = model(user, item)
      train_loss = 0
      for i in range(len(prediction)):
        train_loss += (prediction[i]-label[i])**2
      train_loss = torch.sqrt(train_loss/len(prediction))
      train_loss.backward(retain_graph=True)
```

```
      opt.step()
      train_loss_all += train_loss.item()
    train_loss_all /= len(train_loader)
    if epoch % LR_DECAY_STEP == 0:
        for param_group in opt.param_groups:
            param_group['lr'] = param_group['lr'] / LR_DECAY_VALUE
    print('epoch', epoch,'; rmse train loss', train_loss_all)
    #explicit
    model.eval()
    mape_test_loss_all = 0
    rmse_test_loss_all = 0
    for user, item, label in test_loader:
      prediction = model(user,item)
      rmse_loss = 0
      for i in range(len(prediction)):
        rmse_loss += (prediction[i]-label[i])**2
      rmse_loss = torch.sqrt(rmse_loss/len(prediction))
      rmse_test_loss_all += rmse_loss
      mean_abs_percentage_error = MeanAbsolutePercentageError()
      mape_loss = mean_abs_percentage_error(prediction, label)
      mape_test_loss_all += mape_loss
    rmse_test_loss_all /= len(test_loader)
    mape_test_loss_all /= len(test_loader)

    print('epoch:', epoch, 'rmse test loss:', rmse_test_loss_all, ";mape test␣
↪loss",mape_test_loss_all)
```

```
epoch 1 ; rmse train loss 6.630718482120199
epoch: 1 rmse test loss: tensor(1.0912, grad_fn=<DivBackward0>) ;mape test loss
tensor(0.3403, grad_fn=<DivBackward0>)
epoch 2 ; rmse train loss 5.702629783939515
epoch: 2 rmse test loss: tensor(1.0864, grad_fn=<DivBackward0>) ;mape test loss
tensor(0.3427, grad_fn=<DivBackward0>)
epoch 3 ; rmse train loss 4.524873235016405
epoch: 3 rmse test loss: tensor(1.0900, grad_fn=<DivBackward0>) ;mape test loss
tensor(0.3404, grad_fn=<DivBackward0>)
epoch 4 ; rmse train loss 9.014894124366858
epoch: 4 rmse test loss: tensor(1.0933, grad_fn=<DivBackward0>) ;mape test loss
tensor(0.3396, grad_fn=<DivBackward0>)
epoch 5 ; rmse train loss 5.909568913702751
epoch: 5 rmse test loss: tensor(1.0916, grad_fn=<DivBackward0>) ;mape test loss
tensor(0.3497, grad_fn=<DivBackward0>)
epoch 6 ; rmse train loss 1.0635818597656885
epoch: 6 rmse test loss: tensor(1.0855, grad_fn=<DivBackward0>) ;mape test loss
tensor(0.3436, grad_fn=<DivBackward0>)
epoch 7 ; rmse train loss 1.0618533705425306
epoch: 7 rmse test loss: tensor(1.0859, grad_fn=<DivBackward0>) ;mape test loss
```

```
tensor(0.3437, grad_fn=<DivBackward0>)
epoch 8 ; rmse train loss 1.0619957390340466
epoch: 8 rmse test loss: tensor(1.0959, grad_fn=<DivBackward0>) ;mape test loss
tensor(0.3528, grad_fn=<DivBackward0>)
epoch 9 ; rmse train loss 1.0619348970123275
epoch: 9 rmse test loss: tensor(1.0899, grad_fn=<DivBackward0>) ;mape test loss
tensor(0.3422, grad_fn=<DivBackward0>)
epoch 10 ; rmse train loss 1.0619538120007506
epoch: 10 rmse test loss: tensor(1.0921, grad_fn=<DivBackward0>) ;mape test loss
tensor(0.3452, grad_fn=<DivBackward0>)
epoch 11 ; rmse train loss 1.058583112162417
epoch: 11 rmse test loss: tensor(1.0864, grad_fn=<DivBackward0>) ;mape test loss
tensor(0.3439, grad_fn=<DivBackward0>)
epoch 12 ; rmse train loss 1.0585947605188504
epoch: 12 rmse test loss: tensor(1.0855, grad_fn=<DivBackward0>) ;mape test loss
tensor(0.3439, grad_fn=<DivBackward0>)
epoch 13 ; rmse train loss 1.0585975953752917
epoch: 13 rmse test loss: tensor(1.0864, grad_fn=<DivBackward0>) ;mape test loss
tensor(0.3452, grad_fn=<DivBackward0>)
```

4.0.2 implicit

[ ]:
```
LR = 0.001
EPOCHS = 20
BATCH_SIZE = 101
LR_DECAY_STEP = 5
LR_DECAY_VALUE = 10
```

[ ]:
```
from torch.utils.data import Dataset

class PandasDataset(Dataset):
    def __init__(self, dataframe):
        self.dataframe = dataframe
        self.user_list = list(dataframe['user'])
        self.movie_list = list(dataframe['movie'])
        self.rating_list = list(dataframe['rating'])
        if 'user' in self.user_list:
          self.user_list.remove('user')
        if 'movie' in self.movie_list:
          self.movie_list.remove('movie')
        if 'rating' in self.rating_list:
          self.rating_list.remove('rating')
    def __len__(self):
        return len(self.dataframe)

    def __getitem__(self, idx):
```

```
        user = int(self.user_list[idx])
        movie = int(self.movie_list[idx])
        rating = float(self.rating_list[idx])
        return (
        torch.tensor(user, dtype=torch.long),
        torch.tensor(movie, dtype=torch.long),
        torch.tensor(rating, dtype=torch.float)
        )


train_dataset = PandasDataset(train_df_implicit)
test_dataset = PandasDataset(test_df_implicit)
train_loader = DataLoader(train_dataset, BATCH_SIZE, shuffle=True,␣
 ↪num_workers=2)
test_loader = DataLoader(test_dataset, BATCH_SIZE, shuffle=False, num_workers=2)
```

[ ]:
```
users_train = list(train_df_implicit['user'])
movies_train = list(train_df_implicit['movie'])
model = Model(deepwalk,users_train,movies_train,k=32,batch_size=100)
```

```
ModuleList(
  (0): Linear(in_features=64, out_features=128, bias=True)
  (1): Linear(in_features=128, out_features=48, bias=True)
  (2): Linear(in_features=48, out_features=1, bias=True)
)
```

[ ]:
```
#model.to(device)
model.reset_parameters()
opt = optim.Adam(model.parameters(), lr=LR)
```

[ ]:
```
#evaluation part
def hit(ng_item, pred_items):
    if ng_item in pred_items:
        return 1
    return 0

def ndcg(ng_item, pred_items):
    if ng_item in pred_items:
        index = pred_items.index(ng_item)
        return np.reciprocal(np.log2(index+2))
    return 0
```

[ ]:
```
os.environ['WANDB_CONSOLE'] = 'off'
```

[ ]:
```
EPOCHS = 20
start_time = time.time()
for epoch in range(1, EPOCHS+1):
```

```
    train_loss_all = 0
    count = 0
    for user, item, label in train_loader:
        # if count % 5000 == 0:
        #     print(count)
        #count += 1
        opt.zero_grad()
        prediction = model(user, item)
        loss = mse_loss(prediction, label)
        loss.backward(retain_graph=True)
        opt.step()
        train_loss_all += np.sqrt(loss.item())
    train_loss_all /= len(train_loader)
    if epoch % LR_DECAY_STEP == 0:
        for param_group in opt.param_groups:
            param_group['lr'] = param_group['lr'] / LR_DECAY_VALUE
    print('epoch', epoch,'; train loss', train_loss_all)
    #explicit
    model.eval()
    top_k = 10
    HR, NDCG = [], []
    for user, item, label in test_loader:
      predictions = model(user, item)
      _, indices = torch.topk(predictions, top_k)
      recommends = torch.take(item, indices).cpu().numpy().tolist()
      ng_item = item[0].item() # leave one-out evaluation has only one item per␣
↪user
      HR.append(hit(ng_item, recommends))
      NDCG.append(ndcg(ng_item, recommends))
    HR = np.mean(HR)

    NDCG =np.mean(NDCG)
    print(f'HR for this epoch: {HR}')
    print(f'NDCG for this epoch: {NDCG}')
```

```
epoch 1 ; train loss 2.6841034813983358
HR for this epoch: 0.22081639663365202
NDCG for this epoch: 0.10357380785061919
epoch 2 ; train loss 6.450226109292327
HR for this epoch: 0.03326272399381691
NDCG for this epoch: 0.015387916136252509
epoch 3 ; train loss 3.4800533793467596
HR for this epoch: 0.3269021583557566
NDCG for this epoch: 0.19125780036569368
epoch 4 ; train loss 4.266887818241773
HR for this epoch: 0.05015171466193393
NDCG for this epoch: 0.024821830964859275
```

```
epoch 5 ; train loss 5.540350653120904
HR for this epoch: 0.17232495563061773
NDCG for this epoch: 0.08710721395408408
epoch 6 ; train loss 0.2188565216031923
HR for this epoch: 0.05020896547775806
NDCG for this epoch: 0.024580340909970293
epoch 7 ; train loss 0.18771120433406852
HR for this epoch: 0.20026335375279097
NDCG for this epoch: 0.10215531743846647
epoch 8 ; train loss 0.180393126283129
HR for this epoch: 0.047117421423255285
NDCG for this epoch: 0.023740861513643984
epoch 9 ; train loss 0.18172283984135174
HR for this epoch: 0.035839010705902556
NDCG for this epoch: 0.01648908480906079
epoch 10 ; train loss 0.13170189222693102
HR for this epoch: 0.15526421251502834
NDCG for this epoch: 0.07661231136673914
epoch 11 ; train loss 0.020478245011438096
HR for this epoch: 0.15371844048777694
NDCG for this epoch: 0.07722670618776083
epoch 12 ; train loss 0.012452387609821241
HR for this epoch: 0.04574340184347627
NDCG for this epoch: 0.025543765191009628
epoch 13 ; train loss 0.009205301355547386
HR for this epoch: 0.24285796072594035
NDCG for this epoch: 0.12301294386766848
epoch 14 ; train loss 0.0076093355602286795
HR for this epoch: 0.22310642926661706
NDCG for this epoch: 0.10658346058697701
epoch 15 ; train loss 0.006464872438483494
HR for this epoch: 0.20656094349344478
NDCG for this epoch: 0.09825952082189929
epoch 16 ; train loss 0.0007759700511563884
HR for this epoch: 0.3238106143012538
NDCG for this epoch: 0.18098162509214297
epoch 17 ; train loss 0.0007558331833143808
HR for this epoch: 0.42062174385985
NDCG for this epoch: 0.2659809466675238
```

5 node2vec

[ ]:
```
from torch.utils.data import Dataset

class PandasDataset(Dataset):
    def __init__(self, dataframe):
```

```
        self.dataframe = dataframe
        self.user_list = list(dataframe['user'])
        self.movie_list = list(dataframe['movie'])
        self.rating_list = list(dataframe['rating'])
        if 'user' in self.user_list:
          self.user_list.remove('user')
        if 'movie' in self.movie_list:
          self.movie_list.remove('movie')
        if 'rating' in self.rating_list:
          self.rating_list.remove('rating')
    def __len__(self):
        return len(self.dataframe)

    def __getitem__(self, idx):
      user = int(self.user_list[idx])
      movie = int(self.movie_list[idx])
      rating = float(self.rating_list[idx])
      return (
      torch.tensor(user, dtype=torch.long),
      torch.tensor(movie, dtype=torch.long),
      torch.tensor(rating, dtype=torch.float)
      )
```

[ ]:
```
class Model(torch.nn.Module):
    def __init__(self, embeddings,users,movies,k,batch_size):
        super(Model, self).__init__()
        self.representation = embeddings
        for node in embeddings.keys():
            self.representation.setdefault(node, np.random.random((k,1))/10*np.
 ↪sqrt(k))
        self.k = k
        self.users = users
        self.movies = movies
        self.batch_size = batch_size
        input_size = k*2
        hidden_size1 = [128, 48]
        output_size = 1
        self.model = nn.ModuleList()
        self.model.append(torch.nn.Linear(input_size, hidden_size1[0]))
        #self.model.append(torch.nn.ReLU())
        self.model.append(torch.nn.Linear(hidden_size1[0], hidden_size1[1]))
        self.model.append(torch.nn.ReLU())
        self.model.append(torch.nn.Linear(hidden_size1[1],output_size))
        #self.model.append(torch.nn.Softmax(1))
        print(self.model)
```

```
        self.reset_parameters()
    def reset_parameters(self):
        for i in self.model:
            if isinstance(i,nn.ReLU) or isinstance(i,nn.Softmax):continue
            i.reset_parameters()
    def predict(self,user, movie):
        temp = np.concatenate((self.representation[f'U{user}'],self.
↪representation[f'M{movie}']),axis=0)
        y_pred = self.model(torch.tensor(np.transpose(temp)).float())
        #print(y_pred)
        return y_pred
    def forward(self, user_indices, movie_indices):
        user,movie = user_indices[0], movie_indices[0]
        user_embedding = torch.tensor(self.representation[f'U{user}'])
        movie_embedding = torch.tensor(self.representation[f'M{movie}'])
        embedding_of_batch = torch.cat([user_embedding, movie_embedding],␣
↪dim=-1).reshape((2*self.k,1))
        for i in range(1, len(user_indices)):
            user,movie = user_indices[i], movie_indices[i]
            user_embedding = torch.tensor(self.representation[f'U{user}']).
↪float()
            movie_embedding = torch.tensor(self.representation[f'M{movie}']).
↪float()
            temp = torch.cat([user_embedding, movie_embedding], dim=-1).
↪reshape((2*self.k,1))
            embedding_of_batch = torch.cat([embedding_of_batch, temp], dim=-1)
        embedding_of_batch = torch.t(embedding_of_batch)
        for idx, _ in enumerate(range(len(self.model))):
            embedding_of_batch = self.model[idx](embedding_of_batch)
        result = embedding_of_batch.squeeze()
        return result
```

[ ]:
```
#!pip install stellargraph
```

[ ]:
```
import matplotlib.pyplot as plt

from sklearn.manifold import TSNE
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegressionCV
from sklearn.metrics import accuracy_score

import os
import networkx as nx
import numpy as np
import pandas as pd
#!pip install stellargraph
```

```
from stellargraph.data import BiasedRandomWalk
from stellargraph import StellarGraph
from stellargraph import datasets
from IPython.display import display, HTML

%matplotlib inline
```

[ ]:
```
G = StellarGraph.from_networkx(G_unweighted)
rw = BiasedRandomWalk(G)
walks = rw.run(
    nodes=list(G.nodes()),  # root nodes
    length=40,  # maximum length of a random walk
    n=1,  # number of random walks per root node
    p=0.5,  # Defines (unormalised) probability, 1/p, of returning to source␣
 ↪node
    q=2.0,  # Defines (unormalised) probability, 1/q, for moving away from␣
 ↪source node
)
print("Number of random walks: {}".format(len(walks)))
```

```
      ␣
↪---------------------------------------------------------------------------

       KeyboardInterrupt                         Traceback (most recent call␣
↪last)

       <ipython-input-25-65f4bcc0ebd3> in <module>
         1 G = StellarGraph.from_networkx(G_unweighted)
         2 rw = BiasedRandomWalk(G)
   ----> 3 walks = rw.run(
         4     nodes=list(G.nodes()),  # root nodes
         5     length=40,  # maximum length of a random walk


       /usr/local/lib/python3.8/dist-packages/stellargraph/data/explorer.py in␣
↪run(self, nodes, n, length, p, q, seed, weighted)
       498                     mask = neighbours == previous_node
       499                     weights[mask] *= ip
   --> 500                     mask |= np.isin(neighbours,␣
↪previous_node_neighbours)
       501                     weights[~mask] *= iq
       502


       <__array_function__ internals> in isin(*args, **kwargs)
```

```
       /usr/local/lib/python3.8/dist-packages/numpy/lib/arraysetops.py in␣
↪isin(element, test_elements, assume_unique, invert)
       733     """
       734     element = np.asarray(element)
   --> 735     return in1d(element, test_elements, assume_unique=assume_unique,
       736                 invert=invert).reshape(element.shape)
       737


       <__array_function__ internals> in in1d(*args, **kwargs)


       /usr/local/lib/python3.8/dist-packages/numpy/lib/arraysetops.py in␣
↪in1d(ar1, ar2, assume_unique, invert)
       611     # Otherwise use sorting
       612     if not assume_unique:
   --> 613         ar1, rev_idx = np.unique(ar1, return_inverse=True)
       614         ar2 = np.unique(ar2)
       615


       <__array_function__ internals> in unique(*args, **kwargs)


       /usr/local/lib/python3.8/dist-packages/numpy/lib/arraysetops.py in␣
↪unique(ar, return_index, return_inverse, return_counts, axis)
       270     ar = np.asanyarray(ar)
       271     if axis is None:
   --> 272         ret = _unique1d(ar, return_index, return_inverse,␣
↪return_counts)
       273         return _unpack_tuple(ret)
       274


       /usr/local/lib/python3.8/dist-packages/numpy/lib/arraysetops.py in␣
↪_unique1d(ar, return_index, return_inverse, return_counts)
       346         mask[aux_firstnan + 1:] = False
       347     else:
   --> 348         mask[1:] = aux[1:] != aux[:-1]
       349
       350     ret = (aux[mask],)


       KeyboardInterrupt:
```

[ ]:
```
from gensim.models import Word2Vec
str_walks = [[str(n) for n in walk] for walk in walks]
model = Word2Vec(str_walks, size=32, window=5, min_count=0, sg=1, workers=2,␣
 ↪iter=1)
```

[ ]:
```
model.save("node2vec.model")
```

[ ]:
```
model = Word2Vec.load("node2vec.model")
```

[ ]:
```
G = StellarGraph.from_networkx(G_unweighted)
embeddings= {}
for node in G.nodes():
  embeddings[node] = model.wv.get_vector(node)
```

[ ]:
```

```

5.0.1 explicit

[ ]:
```
train_df = train_df.drop(train_df[train_df['user']=='user'].index)
test_df = test_df.drop(test_df[test_df['user']=='user'].index)
train_dataset = PandasDataset(train_df)
test_dataset = PandasDataset(test_df)
```

[ ]:
```
users_train = list(user_movie_train.keys())
movies_train = list(movie_user_train.keys())
model = Model(embeddings,users_train,movies_train,k=32,batch_size=100)
```

```
ModuleList(
  (0): Linear(in_features=64, out_features=128, bias=True)
  (1): Linear(in_features=128, out_features=48, bias=True)
  (2): ReLU()
  (3): Linear(in_features=48, out_features=1, bias=True)
)
```

[ ]:
```
LR = 0.001
EPOCHS = 20
BATCH_SIZE = 100
LR_DECAY_STEP = 10
LR_DECAY_VALUE = 5
```

[ ]:
```
train_loader = DataLoader(train_dataset, BATCH_SIZE, shuffle=True,␣
 ↪num_workers=2)
test_loader = DataLoader(test_dataset, BATCH_SIZE, shuffle=False, num_workers=2)
```

[ ]:
```
model.reset_parameters()
opt = optim.Adam(model.parameters(), lr=LR)
```

[ ]:
```
EPOCHS = 20
start_time = time.time()
for epoch in range(1, EPOCHS+1):
    model.train()
    train_loss_all = 0
    for user, item, label in train_loader:
      opt.zero_grad()
      prediction = model(user, item)
      rmse_loss = torch.sqrt(mse_loss(prediction, label))
      rmse_loss.backward(retain_graph=True)
      opt.step()
      train_loss_all += rmse_loss
    train_loss_all /= len(train_loader)
    if epoch % LR_DECAY_STEP == 0:
        for param_group in opt.param_groups:
            param_group['lr'] = param_group['lr'] / LR_DECAY_VALUE
    print('epoch', epoch,'; rmse train loss', train_loss_all)
    #explicit
    model.eval()
    count = 0
    rmse_test_loss_all = 0
    mape_test_loss_all = 0
    for user, item, label in test_loader:
        prediction = model(user,item)
        mseloss = mse_loss(prediction,label)
        rmse_test_loss_all += np.sqrt(mseloss.item())

        mean_abs_percentage_error = MeanAbsolutePercentageError()
        mape_loss = mean_abs_percentage_error(prediction, label)
        mape_test_loss_all += mape_loss
        count += 1
    rmse_test_loss_all /= len(test_loader)
    mape_test_loss_all /= len(test_loader)
    print('epoch:', epoch, 'rmse test loss:', rmse_test_loss_all, ";mape test␣
 ↪loss",mape_test_loss_all)
```

```
epoch 1 ; rmse train loss tensor(1.0759, grad_fn=<DivBackward0>)
epoch: 1 rmse test loss: 1.0949085500721705 ;mape test loss tensor(0.3498,
grad_fn=<DivBackward0>)
epoch 2 ; rmse train loss tensor(1.0719, grad_fn=<DivBackward0>)
epoch: 2 rmse test loss: 1.093936159611317 ;mape test loss tensor(0.3464,
grad_fn=<DivBackward0>)
epoch 3 ; rmse train loss tensor(1.0712, grad_fn=<DivBackward0>)
epoch: 3 rmse test loss: 1.0917092623017042 ;mape test loss tensor(0.3476,
grad_fn=<DivBackward0>)
epoch 4 ; rmse train loss tensor(1.0704, grad_fn=<DivBackward0>)
epoch: 4 rmse test loss: 1.099202837189868 ;mape test loss tensor(0.3437,
```

```
grad_fn=<DivBackward0>)
epoch 5 ; rmse train loss tensor(1.0699, grad_fn=<DivBackward0>)
epoch: 5 rmse test loss: 1.099108513001521 ;mape test loss tensor(0.3432,
grad_fn=<DivBackward0>)
epoch 6 ; rmse train loss tensor(1.0696, grad_fn=<DivBackward0>)
epoch: 6 rmse test loss: 1.091149659389651 ;mape test loss tensor(0.3458,
grad_fn=<DivBackward0>)
epoch 7 ; rmse train loss tensor(1.0694, grad_fn=<DivBackward0>)
epoch: 7 rmse test loss: 1.093061747325245 ;mape test loss tensor(0.3508,
grad_fn=<DivBackward0>)
epoch 8 ; rmse train loss tensor(1.0691, grad_fn=<DivBackward0>)
epoch: 8 rmse test loss: 1.0895797485041416 ;mape test loss tensor(0.3476,
grad_fn=<DivBackward0>)
epoch 9 ; rmse train loss tensor(1.0687, grad_fn=<DivBackward0>)
epoch: 9 rmse test loss: 1.0905993155472267 ;mape test loss tensor(0.3452,
grad_fn=<DivBackward0>)
epoch 10 ; rmse train loss tensor(1.0683, grad_fn=<DivBackward0>)
epoch: 10 rmse test loss: 1.09183307235938 ;mape test loss tensor(0.3438,
grad_fn=<DivBackward0>)
epoch 11 ; rmse train loss tensor(1.0662, grad_fn=<DivBackward0>)
epoch: 11 rmse test loss: 1.0885155818498695 ;mape test loss tensor(0.3465,
grad_fn=<DivBackward0>)
epoch 12 ; rmse train loss tensor(1.0659, grad_fn=<DivBackward0>)
epoch: 12 rmse test loss: 1.0894548392580388 ;mape test loss tensor(0.3446,
grad_fn=<DivBackward0>)
epoch 13 ; rmse train loss tensor(1.0656, grad_fn=<DivBackward0>)
epoch: 13 rmse test loss: 1.087778554356807 ;mape test loss tensor(0.3461,
grad_fn=<DivBackward0>)
epoch 14 ; rmse train loss tensor(1.0653, grad_fn=<DivBackward0>)
epoch: 14 rmse test loss: 1.0881765155245555 ;mape test loss tensor(0.3446,
grad_fn=<DivBackward0>)
epoch 15 ; rmse train loss tensor(1.0650, grad_fn=<DivBackward0>)
epoch: 15 rmse test loss: 1.0895512677137447 ;mape test loss tensor(0.3442,
grad_fn=<DivBackward0>)
epoch 16 ; rmse train loss tensor(1.0647, grad_fn=<DivBackward0>)
epoch: 16 rmse test loss: 1.0899230454200401 ;mape test loss tensor(0.3432,
grad_fn=<DivBackward0>)
epoch 17 ; rmse train loss tensor(1.0643, grad_fn=<DivBackward0>)
epoch: 17 rmse test loss: 1.089918953914187 ;mape test loss tensor(0.3425,
grad_fn=<DivBackward0>)
epoch 18 ; rmse train loss tensor(1.0641, grad_fn=<DivBackward0>)
epoch: 18 rmse test loss: 1.0858311590963123 ;mape test loss tensor(0.3460,
grad_fn=<DivBackward0>)
epoch 19 ; rmse train loss tensor(1.0638, grad_fn=<DivBackward0>)
epoch: 19 rmse test loss: 1.086726818948319 ;mape test loss tensor(0.3438,
grad_fn=<DivBackward0>)
epoch 20 ; rmse train loss tensor(1.0635, grad_fn=<DivBackward0>)
epoch: 20 rmse test loss: 1.0862276125714023 ;mape test loss tensor(0.3450,
```

```
grad_fn=<DivBackward0>)
```

5.0.2 implicit

[ ]:
```
train_dataset = PandasDataset(train_df_implicit)
test_dataset = PandasDataset(test_df_implicit)
train_loader = DataLoader(train_dataset, BATCH_SIZE, shuffle=True,␣
↪num_workers=2)
test_loader = DataLoader(test_dataset, BATCH_SIZE, shuffle=False, num_workers=2)
```

[ ]:
```
#evaluation part
def hit(ng_item, pred_items):
        if ng_item in pred_items:
                return 1
        return 0

def ndcg(ng_item, pred_items):
        if ng_item in pred_items:
                index = pred_items.index(ng_item)
                return np.reciprocal(np.log2(index+2))
        return 0

def metrics(model, test_loader, top_k):
        HR, NDCG = [], []

        for user, item, label in test_loader:

                predictions = model(user, item)
                _, indices = torch.topk(predictions, top_k)
                recommends = torch.take(
                                item, indices).cpu().numpy().tolist()

                ng_item = item[0].item() # leave one-out evaluation has only␣
↪one item per user
                HR.append(hit(ng_item, recommends))
                NDCG.append(ndcg(ng_item, recommends))

        return np.mean(HR), np.mean(NDCG)
```

[ ]:
```
LR = 0.001
EPOCHS = 20
BATCH_SIZE = 100
LR_DECAY_STEP = 10
LR_DECAY_VALUE = 5
```

[ ]:
```
train_loader = DataLoader(train_dataset, BATCH_SIZE, shuffle=True,␣
↪num_workers=2)
test_loader = DataLoader(test_dataset, BATCH_SIZE, shuffle=False, num_workers=2)
```

[ ]:
```
users_train = list(train_df_implicit['user'])
movies_train = list(train_df_implicit['movie'])
model = Model(embeddings,users_train,movies_train,k=32,batch_size=100)
```

```
ModuleList(
  (0): Linear(in_features=64, out_features=128, bias=True)
  (1): Linear(in_features=128, out_features=48, bias=True)
  (2): ReLU()
  (3): Linear(in_features=48, out_features=1, bias=True)
)
```

[ ]:
```
model.reset_parameters()
opt = optim.Adam(model.parameters(), lr=LR)
```

[ ]:
```
EPOCHS = 50
start_time = time.time()
loss_function = nn.BCELoss()
for epoch in range(1, EPOCHS+1):
    train_loss_all = 0
    count = 0
    for user, item, label in train_loader:
        opt.zero_grad()
        prediction = model(user, item)
        loss = torch.sqrt(mse_loss(prediction,label))
        loss.backward(retain_graph=True)
        opt.step()
        train_loss_all += loss.item()
        #print(train_loss_all)
        count += 1
        if count % 5000 == 0:
           break
    train_loss_all /= len(train_loader)
    if epoch % LR_DECAY_STEP == 0:
        for param_group in opt.param_groups:
            param_group['lr'] = param_group['lr'] / LR_DECAY_VALUE
    print('epoch', epoch,'; train loss', train_loss_all)
    #explicit
    model.eval()
    top_k = 10
    HR, NDCG = [], []
    count = 0
    for user, item, label in test_loader:
      prediction = model(user, item)
```

```
      ng_item = item[0].item()# leave one-out evaluation has only one item per␣
↪user
      _, indices = torch.topk(prediction, top_k)

      recommends = torch.take(item, indices).cpu().numpy().tolist()
      # if ng_item in recommends:
      #   HR.append(1)
      # else:
      #   HR.append(0)
      HR.append(hit(ng_item, recommends))
      NDCG.append(ndcg(ng_item, recommends))
      count += 1
    HR = np.mean(HR)

    NDCG =np.mean(NDCG)
    print(f'HR for this epoch: {HR}')
    print(f'NDCG for this epoch: {NDCG}')
```

```
epoch 1 ; train loss 0.25268171003657425
HR for this epoch: 0.10089559006915316
NDCG for this epoch: 0.04655415893525405
epoch 2 ; train loss 0.25201040713602035
```

```
      ␣
↪---------------------------------------------------------------------------

       KeyboardInterrupt                         Traceback (most recent call␣
↪last)

       <ipython-input-48-a5e4d76f8083> in <module>
        27     count = 0
        28     for user, item, label in test_loader:
   ---> 29       prediction = model(user, item)
        30       ng_item = item[0].item()# leave one-out evaluation has only␣
↪one item per user
        31       _, indices = torch.topk(prediction, top_k)


       /usr/local/lib/python3.8/dist-packages/torch/nn/modules/module.py in␣
↪_call_impl(self, *input, **kwargs)
      1188         if not (self._backward_hooks or self._forward_hooks or self.
↪_forward_pre_hooks or _global_backward_hooks
      1189                 or _global_forward_hooks or␣
↪_global_forward_pre_hooks):
   -> 1190             return forward_call(*input, **kwargs)
      1191         # Do not call functions when jit is used
```

```
      1192         full_backward_hooks, non_full_backward_hooks = [], []


       <ipython-input-40-d0f08ada9a28> in forward(self, user_indices,␣
↪movie_indices)
        39             user_embedding = torch.tensor(self.
↪representation[f'U{user}']).float()
        40             movie_embedding = torch.tensor(self.
↪representation[f'M{movie}']).float()
   ---> 41             temp = torch.cat([user_embedding, movie_embedding],␣
↪dim=-1).reshape((2*self.k,1))
        42             embedding_of_batch = torch.cat([embedding_of_batch,␣
↪temp], dim=-1)
        43         embedding_of_batch = torch.t(embedding_of_batch)


       KeyboardInterrupt:
```

[ ]:
```
    # model.eval()
    # top_k = 10
    # HR, NDCG = metrics(model, test_loader, top_k)

    # elapsed_time = time.time() - start_time
    # print("The time elapse of epoch {:03d}".format(epoch) + " is: " +
    #     time.strftime("%H: %M: %S", time.gmtime(elapsed_time)))
    # print("HR: {:.3f}\tNDCG: {:.3f}".format(np.mean(HR), np.mean(NDCG)))

    # if HR > best_hr:
    #   best_hr, best_ndcg, best_epoch = HR, NDCG, epoch
    #   if not os.path.exists(MODEL_PATH):
    #     os.mkdir(MODEL_PATH)
```

[ ]:
```

```

6 deepwalk_advanced

[ ]:
```
from torch.utils.data import Dataset

class PandasDataset(Dataset):
    def __init__(self, dataframe):
        self.dataframe = dataframe
        self.user_list = list(dataframe['user'])
        self.movie_list = list(dataframe['movie'])
        self.rating_list = list(dataframe['rating'])
```

```
        if 'user' in self.user_list:
          self.user_list.remove('user')
        if 'movie' in self.movie_list:
          self.movie_list.remove('movie')
        if 'rating' in self.rating_list:
          self.rating_list.remove('rating')
    def __len__(self):
        return len(self.dataframe)

    def __getitem__(self, idx):
      user = int(self.user_list[idx])
      movie = int(self.movie_list[idx])
      rating = float(self.rating_list[idx])
      return (
      torch.tensor(user, dtype=torch.long),
      torch.tensor(movie, dtype=torch.long),
      torch.tensor(rating, dtype=torch.float)
      )
```

[ ]:
```
class DeepWalk_advanced:
  def __init__(self, G):
    self.G = G
    self.nodes = list(G.nodes())
    self.representation = {}
    self.degree_centralities = nx.degree_centrality(G)
    k = 32 #representation dimention
    w = 5 #window size
    step = 1 #run how many times for each node
    learning_rate = 0.05
    walk_length = 80
    self.workers = multiprocessing.cpu_count()
    self.model = self.DeepWalk(w, k, step, walk_length, learning_rate)
    self.get_representation()
  def RandomWalk_with_weight(self,start_node,walk_length):
    walk = [start_node]
    while len(walk) < walk_length:
          cur = walk[-1]
          cur_nbrs = list(self.G.neighbors(cur))
          possible_node = -1
          highest_degree = -1
          for node in cur_nbrs:
              if self.degree_centralities[node] > highest_degree:
                possible_node = node
                highest_degree = self.degree_centralities[node]
```

```
          if len(cur_nbrs) > 0:
            walk.append(possible_node)
          else:
              break
    return walk

  def get_representation(self):
    for node in self.nodes:
      self.representation[node] = self.model.wv.get_vector(node)
  def DeepWalk(self, w, k, step, walk_length, learning_rate):
    # for node in self.G.nodes:
    #   self.representation.setdefault(node, np.random.random((k,1)))
    walks = []
    random.shuffle(list(self.G.nodes))
    count = 0
    for start_node in self.G.nodes:
      count += 1
      if count % 1000 == 0:
        print(count)
      walk = self.RandomWalk_with_weight(start_node, walk_length)
      walks.append(walk)
    #self.learn_representation(walk, k, w, learning_rate)
    self.model = Word2Vec(walks,window=w,min_count=1,workers=self.
↪workers,size=k,iter=step,sg=1)
    return self.model

deepwalk = DeepWalk_advanced(G_unweighted)
```

```
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
14000
15000
16000
17000
18000
19000
```

```
20000
21000
22000
23000
24000
25000
26000
27000
28000
29000
30000
31000
32000
33000
34000
35000
36000
37000
38000
39000
40000
41000
```

[ ]:
```
deepwalk.model.save('deepwalk_weight.model')
```

[ ]:
```
model = Word2Vec.load('deepwalk_weight.model')
```

[ ]:
```
representations={}
for node in G_unweighted.nodes():
  representations[node] =model.wv.get_vector(node)
```

[ ]:
```
class Model(torch.nn.Module):
    def __init__(self, embeddings,users,movies,k,batch_size):
        super(Model, self).__init__()
        self.representation = embeddings
        for node in embeddings.keys():
            self.representation.setdefault(node, np.random.random((k,1))/10*np.
 ↪sqrt(k))
        self.k = k
        self.users = users
        self.movies = movies
        self.batch_size = batch_size
        input_size = k*2
        hidden_size1 = [128, 48]
        output_size = 1
        self.model = nn.ModuleList()
        self.model.append(torch.nn.Linear(input_size, hidden_size1[0]))
```

```
        #self.model.append(torch.nn.ReLU())
        self.model.append(torch.nn.Linear(hidden_size1[0], hidden_size1[1]))
        self.model.append(torch.nn.ReLU())
        self.model.append(torch.nn.Linear(hidden_size1[1],output_size))
        #self.model.append(torch.nn.Softmax(1))
        print(self.model)
        self.reset_parameters()
    def reset_parameters(self):
        for i in self.model:
            if isinstance(i,nn.ReLU) or isinstance(i,nn.Softmax):continue
            i.reset_parameters()
    def predict(self,user, movie):
        temp = np.concatenate((self.representation[f'U{user}'],self.
↪representation[f'M{movie}']),axis=0)
        y_pred = self.model(torch.tensor(np.transpose(temp)).float())
        #print(y_pred)
        return y_pred
    def forward(self, user_indices, movie_indices):
        user,movie = user_indices[0], movie_indices[0]
        user_embedding = torch.tensor(self.representation[f'U{user}'])
        movie_embedding = torch.tensor(self.representation[f'M{movie}'])
        embedding_of_batch = torch.cat([user_embedding, movie_embedding],␣
↪dim=-1).reshape((2*self.k,1))
        for i in range(1, len(user_indices)):
            user,movie = user_indices[i], movie_indices[i]
            user_embedding = torch.tensor(self.representation[f'U{user}']).
↪float()
            movie_embedding = torch.tensor(self.representation[f'M{movie}']).
↪float()
            temp = torch.cat([user_embedding, movie_embedding], dim=-1).
↪reshape((2*self.k,1))
            embedding_of_batch = torch.cat([embedding_of_batch, temp], dim=-1)
        embedding_of_batch = torch.t(embedding_of_batch)
        for idx, _ in enumerate(range(len(self.model))):
            embedding_of_batch = self.model[idx](embedding_of_batch)
        result = embedding_of_batch.squeeze()
        return result
```

[ ]:

6.0.1 explicit

[ ]:
```
train_df = train_df.drop(train_df[train_df['user']=='user'].index)
test_df = test_df.drop(test_df[test_df['user']=='user'].index)
train_dataset = PandasDataset(train_df)
```

```
test_dataset = PandasDataset(test_df)
```

[ ]:
```
train_loader = DataLoader(train_dataset, BATCH_SIZE, shuffle=True,␣
↪num_workers=2)
test_loader = DataLoader(test_dataset, BATCH_SIZE, shuffle=False, num_workers=2)
```

[ ]:
```
users_train = list(user_movie_train.keys())
movies_train = list(movie_user_train.keys())
model = Model(representations,users_train,movies_train,k=32,batch_size=100)
```

```
ModuleList(
  (0): Linear(in_features=64, out_features=128, bias=True)
  (1): Linear(in_features=128, out_features=48, bias=True)
  (2): ReLU()
  (3): Linear(in_features=48, out_features=1, bias=True)
)
```

[ ]:
```
LR = 0.05
EPOCHS = 20
BATCH_SIZE = 100
LR_DECAY_STEP = 10
LR_DECAY_VALUE = 5
```

[ ]:
```
model.reset_parameters()
opt = optim.Adam(model.parameters(), lr=LR)
```

[ ]:
```
EPOCHS = 20
start_time = time.time()
for epoch in range(1, EPOCHS+1):
    model.train()
    train_loss_all = 0
    count = 0
    for user,item, label in train_loader:
      opt.zero_grad()
      prediction = model(user, item)
      rmse_loss = torch.sqrt(mse_loss(prediction, label))
      rmse_loss.backward(retain_graph=True)
      opt.step()
      train_loss_all += rmse_loss
      count += 1
    train_loss_all /= len(train_loader)

    if epoch % LR_DECAY_STEP == 0:
        for param_group in opt.param_groups:
            param_group['lr'] = param_group['lr'] / LR_DECAY_VALUE
    print('epoch', epoch,'; rmse train loss', train_loss_all)
    #explicit
```

```
    model.eval()
    count = 0
    rmse_test_loss_all = 0
    mape_test_loss_all = 0
    for user,item, label in test_loader:
        prediction = model(user,item)
        mseloss = mse_loss(prediction,label)
        rmse_test_loss_all += np.sqrt(mseloss.item())

        mean_abs_percentage_error = MeanAbsolutePercentageError()
        mape_loss = mean_abs_percentage_error(prediction, label)
        mape_test_loss_all += mape_loss
        count += 1
    rmse_test_loss_all /= len(test_loader)
    mape_test_loss_all /= len(test_loader)
    print('epoch:', epoch, 'rmse test loss:', rmse_test_loss_all, ";mape test␣
↪loss",mape_test_loss_all)
```

```
epoch 1 ; rmse train loss tensor(1.0806, grad_fn=<DivBackward0>)
epoch: 1 rmse test loss: 1.096405509222969 ;mape test loss tensor(0.3492,
grad_fn=<DivBackward0>)
epoch 2 ; rmse train loss tensor(1.0789, grad_fn=<DivBackward0>)
epoch: 2 rmse test loss: 1.103637070384907 ;mape test loss tensor(0.3464,
grad_fn=<DivBackward0>)
epoch 3 ; rmse train loss tensor(1.0789, grad_fn=<DivBackward0>)
epoch: 3 rmse test loss: 1.1044841727935812 ;mape test loss tensor(0.3461,
grad_fn=<DivBackward0>)
epoch 4 ; rmse train loss tensor(1.0789, grad_fn=<DivBackward0>)
epoch: 4 rmse test loss: 1.094616337737437 ;mape test loss tensor(0.3503,
grad_fn=<DivBackward0>)
epoch 5 ; rmse train loss tensor(1.0789, grad_fn=<DivBackward0>)
epoch: 5 rmse test loss: 1.1003259243621017 ;mape test loss tensor(0.3475,
grad_fn=<DivBackward0>)
epoch 6 ; rmse train loss tensor(1.0789, grad_fn=<DivBackward0>)
epoch: 6 rmse test loss: 1.1009908411039304 ;mape test loss tensor(0.3473,
grad_fn=<DivBackward0>)
epoch 7 ; rmse train loss tensor(1.0789, grad_fn=<DivBackward0>)
```

```
      ␣
↪---------------------------------------------------------------------------

       KeyboardInterrupt                         Traceback (most recent call␣
↪last)

       <ipython-input-27-8e64b78a55c5> in <module>
        25     mape_test_loss_all = 0
        26     for user,item, label in test_loader:
```

```
---> 27         prediction = model(user,item)
     28         mseloss = mse_loss(prediction,label)
     29         rmse_test_loss_all += np.sqrt(mseloss.item())


/usr/local/lib/python3.8/dist-packages/torch/nn/modules/module.py in␣
↪_call_impl(self, *input, **kwargs)
   1188         if not (self._backward_hooks or self._forward_hooks or self.
↪_forward_pre_hooks or _global_backward_hooks
   1189                 or _global_forward_hooks or␣
↪_global_forward_pre_hooks):
-> 1190             return forward_call(*input, **kwargs)
   1191         # Do not call functions when jit is used
   1192         full_backward_hooks, non_full_backward_hooks = [], []


<ipython-input-22-95e90570e653> in forward(self, user_indices,␣
↪movie_indices)
     39             user_embedding = torch.tensor(self.
↪representation[f'U{user}']).float()
     40             movie_embedding = torch.tensor(self.
↪representation[f'M{movie}']).float()
---> 41             temp = torch.cat([user_embedding, movie_embedding],␣
↪dim=-1).reshape((2*self.k,1))
     42             embedding_of_batch = torch.cat([embedding_of_batch,␣
↪temp], dim=-1)
     43         embedding_of_batch = torch.t(embedding_of_batch)


KeyboardInterrupt:
```

[ ]:
```
test_df.head()
```

[ ]:

	movie	user	rating	date
0	14725	40563	4	2005-08-06
1	14856	89321	5	2005-08-06
2	15788	123124	2	2005-08-06
3	788	82715	3	2005-08-06
4	13981	72036	4	2005-08-06

6.0.2 implicit

[ ]:
```
LR = 0.05
EPOCHS = 20
BATCH_SIZE = 100
LR_DECAY_STEP = 20
LR_DECAY_VALUE = 5
train_dataset = PandasDataset(train_df_implicit)
test_dataset = PandasDataset(test_df_implicit)
train_loader = DataLoader(train_dataset, BATCH_SIZE, shuffle=True,␣
↪num_workers=2)
test_loader = DataLoader(test_dataset, BATCH_SIZE, shuffle=False, num_workers=2)
users_train = list(user_movie_train.keys())
movies_train = list(movie_user_train.keys())
model = Model(representations,users_train,movies_train,k=32,batch_size=100)
model.reset_parameters()
opt = optim.Adam(model.parameters(), lr=LR)
```

[ ]:
```
EPOCHS = 50
start_time = time.time()
loss_function = nn.BCELoss()
for epoch in range(1, EPOCHS+1):
    train_loss_all = 0
    count = 0
    for user, item, label in train_loader:
        opt.zero_grad()
        prediction = model(user, item)
        loss = torch.sqrt(mse_loss(prediction,label))
        loss.backward(retain_graph=True)
        opt.step()
        train_loss_all += loss.item()
        #print(train_loss_all)
        count += 1
        if count % 5000 == 0:
           break
    train_loss_all /= len(train_loader)
    if epoch % LR_DECAY_STEP == 0:
        for param_group in opt.param_groups:
            param_group['lr'] = param_group['lr'] / LR_DECAY_VALUE
    print('epoch', epoch,'; train loss', train_loss_all)
    #explicit
    model.eval()
    top_k = 10
    HR, NDCG = [], []
    count = 0
    for user, item, label in test_loader:
      predictions = model(user, item)
```

```
      _, indices = torch.topk(predictions, top_k)
      recommends = torch.take(item, indices).cpu().numpy().tolist()
      ng_item = item[0].item() # leave one-out evaluation has only one item per␣
↪user
      HR.append(hit(ng_item, recommends))
      NDCG.append(ndcg(ng_item, recommends))
      HR.append(hit(ng_item, recommends))
      NDCG.append(ndcg(ng_item, recommends))
      count += 1
    HR = np.mean(HR)

    NDCG =np.mean(NDCG)
    print(f'HR for this epoch: {HR}')
    print(f'NDCG for this epoch: {NDCG}')
```

```
      ␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-17-a878e5405d73> in <module>
         5     train_loss_all = 0
         6     count = 0
    ----> 7     for user, item, label in train_loader:
         8         opt.zero_grad()
         9         prediction = model(user, item)


       NameError: name 'train_loader' is not defined
```

7 deep walk advanced neural network (choose lowest degree centrality)

[ ]:
```
import networkx as nx
```

[ ]:
```
class DeepWalk_advanced:
  def __init__(self, G):
    self.G = G
    self.nodes = list(G.nodes())
    self.representation = {}
    self.degree_centralities = nx.degree_centrality(G)
    k = 32 #representation dimention
```

```
    w = 5 #window size
    step = 1 #run how many times for each node
    learning_rate = 0.05
    walk_length = 80
    self.workers = multiprocessing.cpu_count()
    self.model = self.DeepWalk(w, k, step, walk_length, learning_rate)
    self.get_representation()
  def RandomWalk_with_weight(self,start_node,walk_length):
    walk = [start_node]
    while len(walk) < walk_length:
          cur = walk[-1]
          cur_nbrs = list(self.G.neighbors(cur))
          possible_node = -1
          lowest_degree = len(self.G.nodes())
          for node in cur_nbrs:
              if self.degree_centralities[node] <lowest_degree:
                possible_node = node
                lowest_degree = self.degree_centralities[node]

          if len(cur_nbrs) > 0:
            walk.append(possible_node)
          else:
              break
    return walk

  def get_representation(self):
    for node in self.nodes:
      self.representation[node] = self.model.wv.get_vector(node)
  def DeepWalk(self, w, k, step, walk_length, learning_rate):
    # for node in self.G.nodes:
    #   self.representation.setdefault(node, np.random.random((k,1)))
    walks = []
    random.shuffle(list(self.G.nodes))
    count = 0
    for start_node in self.G.nodes:
      count += 1
      if count % 1000 == 0:
        print(count)
      walk = self.RandomWalk_with_weight(start_node, walk_length)
      walks.append(walk)
    #self.learn_representation(walk, k, w, learning_rate)
    self.model = Word2Vec(walks,window=w,min_count=1,workers=self.
↪workers,size=k,iter=step,sg=1)
    return self.model

deepwalk = DeepWalk_advanced(G_unweighted)
```

```
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
14000
15000
16000
17000
18000
19000
20000
21000
22000
23000
24000
25000
26000
27000
28000
29000
30000
31000
32000
33000
34000
35000
36000
37000
38000
39000
40000
41000
```

[ ]:
```
deepwalk.model.save('deepwalk_weight2.model')
```

[ ]:
```
model_embedding = Word2Vec.load('deepwalk_weight2.model')
```

[ ]:
```
representations={}
for node in G_unweighted.nodes():
  representations[node] =model_embedding.wv.get_vector(node)
```

[ ]:
```
LR = 0.01
EPOCHS = 20
BATCH_SIZE = 100
LR_DECAY_STEP = 5
LR_DECAY_VALUE = 10
```

[ ]:
```

```

[ ]:
```
users_train = list(user_movie_train.keys())
movies_train = list(movie_user_train.keys())
model = Model(representations,users_train,movies_train,k=32,batch_size=100)
model.reset_parameters()
opt = optim.Adam(model.parameters(), lr=LR)
```

```
ModuleList(
  (0): Linear(in_features=64, out_features=128, bias=True)
  (1): Linear(in_features=128, out_features=48, bias=True)
  (2): ReLU()
  (3): Linear(in_features=48, out_features=1, bias=True)
)
```

[ ]:
```
EPOCHS = 20
start_time = time.time()
for epoch in range(1, EPOCHS+1):
    model.train()
    train_loss_all = 0
    count = 0
    for user,item, label in train_loader:
      opt.zero_grad()
      prediction = model(user, item)
      rmse_loss = torch.sqrt(mse_loss(prediction, label))
      rmse_loss.backward(retain_graph=True)
      opt.step()
      train_loss_all += rmse_loss
      count += 1
    train_loss_all /= len(train_loader)

    if epoch % LR_DECAY_STEP == 0:
        for param_group in opt.param_groups:
            param_group['lr'] = param_group['lr'] / LR_DECAY_VALUE
    print('epoch', epoch,'; rmse train loss', train_loss_all)
    #explicit
    model.eval()
```

```
    count = 0
    rmse_test_loss_all = 0
    mape_test_loss_all = 0
    for user,item, label in test_loader:
        prediction = model(user,item)
        mseloss = mse_loss(prediction,label)
        rmse_test_loss_all += np.sqrt(mseloss.item())

        mean_abs_percentage_error = MeanAbsolutePercentageError()
        mape_loss = mean_abs_percentage_error(prediction, label)
        mape_test_loss_all += mape_loss
        count += 1
    rmse_test_loss_all /= len(test_loader)
    mape_test_loss_all /= len(test_loader)
    print('epoch:', epoch, 'rmse test loss:', rmse_test_loss_all, ";mape test␣
↪loss",mape_test_loss_all)
```

```
epoch 1 ; rmse train loss tensor(1.0800, grad_fn=<DivBackward0>)
epoch: 1 rmse test loss: 1.098472362056514 ;mape test loss tensor(0.3482,
grad_fn=<DivBackward0>)
epoch 2 ; rmse train loss tensor(1.0780, grad_fn=<DivBackward0>)
epoch: 2 rmse test loss: 1.0968408464569934 ;mape test loss tensor(0.3490,
grad_fn=<DivBackward0>)
epoch 3 ; rmse train loss tensor(1.0780, grad_fn=<DivBackward0>)
epoch: 3 rmse test loss: 1.099464709778565 ;mape test loss tensor(0.3478,
grad_fn=<DivBackward0>)
epoch 4 ; rmse train loss tensor(1.0780, grad_fn=<DivBackward0>)
epoch: 4 rmse test loss: 1.0944098028061822 ;mape test loss tensor(0.3505,
grad_fn=<DivBackward0>)
epoch 5 ; rmse train loss tensor(1.0780, grad_fn=<DivBackward0>)
epoch: 5 rmse test loss: 1.0989571464728416 ;mape test loss tensor(0.3480,
grad_fn=<DivBackward0>)
epoch 6 ; rmse train loss tensor(1.0778, grad_fn=<DivBackward0>)
epoch: 6 rmse test loss: 1.0974252633503654 ;mape test loss tensor(0.3487,
grad_fn=<DivBackward0>)
epoch 7 ; rmse train loss tensor(1.0778, grad_fn=<DivBackward0>)
epoch: 7 rmse test loss: 1.0983662294197194 ;mape test loss tensor(0.3483,
grad_fn=<DivBackward0>)
epoch 8 ; rmse train loss tensor(1.0778, grad_fn=<DivBackward0>)
epoch: 8 rmse test loss: 1.0975023986389487 ;mape test loss tensor(0.3487,
grad_fn=<DivBackward0>)
epoch 9 ; rmse train loss tensor(1.0778, grad_fn=<DivBackward0>)
epoch: 9 rmse test loss: 1.0965029306230714 ;mape test loss tensor(0.3492,
grad_fn=<DivBackward0>)
epoch 10 ; rmse train loss tensor(1.0778, grad_fn=<DivBackward0>)
epoch: 10 rmse test loss: 1.0973575303683658 ;mape test loss tensor(0.3487,
grad_fn=<DivBackward0>)
```

```
epoch 11 ; rmse train loss tensor(1.0777, grad_fn=<DivBackward0>)
epoch: 11 rmse test loss: 1.0976251480891916 ;mape test loss tensor(0.3486,
grad_fn=<DivBackward0>)


      ␣
↪------------------------------------------------------------------------

       KeyboardInterrupt                         Traceback (most recent call␣
↪last)

       <ipython-input-40-8e64b78a55c5> in <module>
        10       rmse_loss = torch.sqrt(mse_loss(prediction, label))
        11       rmse_loss.backward(retain_graph=True)
   ---> 12       opt.step()
        13       train_loss_all += rmse_loss
        14       count += 1


       /usr/local/lib/python3.8/dist-packages/torch/optim/optimizer.py in␣
↪wrapper(*args, **kwargs)
       138                 profile_name = "Optimizer.step#{}.step".format(obj.
↪__class__.__name__)
       139                 with torch.autograd.profiler.
↪record_function(profile_name):
   --> 140                     out = func(*args, **kwargs)
       141                     obj._optimizer_step_code()
       142                     return out


       /usr/local/lib/python3.8/dist-packages/torch/optim/optimizer.py in␣
↪_use_grad(self, *args, **kwargs)
        21         try:
        22             torch.set_grad_enabled(self.defaults['differentiable'])
   ---> 23             ret = func(self, *args, **kwargs)
        24         finally:
        25             torch.set_grad_enabled(prev_grad)


       /usr/local/lib/python3.8/dist-packages/torch/optim/adam.py in step(self,␣
↪closure, grad_scaler)
       232                     state_steps.append(state['step'])
       233
   --> 234             adam(params_with_grad,
       235                  grads,
       236                  exp_avgs,
```

```
       /usr/local/lib/python3.8/dist-packages/torch/optim/adam.py in␣
↪adam(params, grads, exp_avgs, exp_avg_sqs, max_exp_avg_sqs, state_steps,␣
↪foreach, capturable, differentiable, fused, grad_scale, found_inf, amsgrad,␣
↪beta1, beta2, lr, weight_decay, eps, maximize)
       298         func = _single_tensor_adam
       299
   --> 300     func(params,
       301          grads,
       302          exp_avgs,


       /usr/local/lib/python3.8/dist-packages/torch/optim/adam.py in␣
↪_single_tensor_adam(params, grads, exp_avgs, exp_avg_sqs, max_exp_avg_sqs,␣
↪state_steps, grad_scale, found_inf, amsgrad, beta1, beta2, lr, weight_decay,␣
↪eps, maximize, capturable, differentiable)
       408                 denom = (max_exp_avg_sqs[i].sqrt() /␣
↪bias_correction2_sqrt).add_(eps)
       409             else:
   --> 410                 denom = (exp_avg_sq.sqrt() / bias_correction2_sqrt).
↪add_(eps)
       411
       412             param.addcdiv_(exp_avg, denom, value=-step_size)


       KeyboardInterrupt:
```

7.1 implicit

[ ]:
```
class Model(torch.nn.Module):
    def __init__(self, embeddings,users,movies,k,batch_size):
        super(Model, self).__init__()
        self.representation = embeddings
        for node in embeddings.keys():
            self.representation.setdefault(node, np.random.random((k,1))/10*np.
 ↪sqrt(k))
        self.k = k
        self.users = users
        self.movies = movies
        self.batch_size = batch_size
        input_size = k*2
        hidden_size1 = [128, 48]
        output_size = 1
        self.model = nn.ModuleList()
        self.model.append(torch.nn.Linear(input_size, hidden_size1[0]))
        #self.model.append(torch.nn.ReLU())
```

```
        self.model.append(torch.nn.Linear(hidden_size1[0], hidden_size1[1]))
        #self.model.append(torch.nn.ReLU())
        self.model.append(torch.nn.Linear(hidden_size1[1],output_size))
        #self.model.append(torch.nn.Softmax(1))
        print(self.model)
        self.reset_parameters()
    def reset_parameters(self):
        for i in self.model:
            if isinstance(i,nn.ReLU) or isinstance(i,nn.Softmax):continue
            i.reset_parameters()
    def predict(self,user, movie):
        temp = np.concatenate((self.representation[f'U{user}'],self.
↪representation[f'M{movie}']),axis=0)
        y_pred = self.model(torch.tensor(np.transpose(temp)).float())
        #print(y_pred)
        return y_pred
    def forward(self, user_indices, movie_indices):
        user,movie = user_indices[0], movie_indices[0]
        user_embedding = torch.tensor(self.representation[f'U{user}'])
        movie_embedding = torch.tensor(self.representation[f'M{movie}'])
        embedding_of_batch = torch.cat([user_embedding, movie_embedding],␣
↪dim=-1).reshape((2*self.k,1))
        for i in range(1, len(user_indices)):
            user,movie = user_indices[i], movie_indices[i]
            user_embedding = torch.tensor(self.representation[f'U{user}']).
↪float()
            movie_embedding = torch.tensor(self.representation[f'M{movie}']).
↪float()
            temp = torch.cat([user_embedding, movie_embedding], dim=-1).
↪reshape((2*self.k,1))
            embedding_of_batch = torch.cat([embedding_of_batch, temp], dim=-1)

        embedding_of_batch = torch.t(embedding_of_batch)
        embedding_of_batch = self.model[0](embedding_of_batch)
        embedding_of_batch = self.model[1](embedding_of_batch)
        embedding_of_batch = self.model[2](embedding_of_batch)
        result = embedding_of_batch.squeeze()
        return result
```

[ ]:
```
LR = 0.001
EPOCHS = 20
BATCH_SIZE = 100
LR_DECAY_STEP = 20
LR_DECAY_VALUE = 5
train_dataset = PandasDataset(train_df_implicit)
test_dataset = PandasDataset(test_df_implicit)
```

```
train_loader = DataLoader(train_dataset, BATCH_SIZE, shuffle=True,␣
↪num_workers=2)
test_loader = DataLoader(test_dataset, BATCH_SIZE, shuffle=False, num_workers=2)
users_train = list(user_movie_train.keys())
movies_train = list(movie_user_train.keys())
model = Model(representations,users_train,movies_train,k=32,batch_size=100)
model.reset_parameters()
opt = optim.Adam(model.parameters(), lr=LR)
```

```
      ␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-1-70f799a2eeba> in <module>
         4 LR_DECAY_STEP = 20
         5 LR_DECAY_VALUE = 5
   ----> 6 train_dataset = PandasDataset(train_df_implicit)
         7 test_dataset = PandasDataset(test_df_implicit)
         8 train_loader = DataLoader(train_dataset, BATCH_SIZE, shuffle=True,␣
↪num_workers=2)

       NameError: name 'PandasDataset' is not defined
```

[ ]:
```
#evaluation part
def hit(ng_item, pred_items):
    if ng_item in pred_items:
        return 1
    return 0

def ndcg(ng_item, pred_items):
    if ng_item in pred_items:
        index = pred_items.index(ng_item)
        return np.reciprocal(np.log2(index+2))
    return 0
```

[ ]:
```
EPOCHS = 20
start_time = time.time()
loss_function = nn.BCELoss()
for epoch in range(1, EPOCHS+1):
    train_loss_all = 0
    count = 0
    for user, item, label in train_loader:
```

```
        opt.zero_grad()
        prediction = model(user, item)
        #print(prediction)
        loss = torch.sqrt(mse_loss(prediction,label))
        loss.backward(retain_graph=True)
        opt.step()
        train_loss_all += loss.item()
        #print(train_loss_all)
        count += 1
    train_loss_all /= len(train_loader)
    if epoch % LR_DECAY_STEP == 0:
        for param_group in opt.param_groups:
            param_group['lr'] = param_group['lr'] / LR_DECAY_VALUE
    print('epoch', epoch,'; train loss', train_loss_all)
    #explicit
    model.eval()
    top_k = 10
    HR, NDCG = [], []
    count = 0
    for user, item, label in test_loader:
      predictions = model(user, item)
      _, indices = torch.topk(predictions, top_k)
      recommends = torch.take(item, indices).cpu().numpy().tolist()
      ng_item = item[0].item() # leave one-out evaluation has only one item per␣
↪user
      HR.append(hit(ng_item, recommends))
      NDCG.append(ndcg(ng_item, recommends))
      HR.append(hit(ng_item, recommends))
      NDCG.append(ndcg(ng_item, recommends))
      count += 1
    HR = np.mean(HR)

    NDCG =np.mean(NDCG)
    print(f'HR for this epoch: {HR}')
    print(f'NDCG for this epoch: {NDCG}')
```

```
Exception ignored in: <function _MultiProcessingDataLoaderIter.__del__ at
0x7f847b973790>
Traceback (most recent call last):
  File "/usr/local/lib/python3.8/dist-packages/torch/utils/data/dataloader.py",
line 1466, in __del__
    self._shutdown_workers()
  File "/usr/local/lib/python3.8/dist-packages/torch/utils/data/dataloader.py",
line 1449, in _shutdown_workers
    if w.is_alive():
  File "/usr/lib/python3.8/multiprocessing/process.py", line 160, in is_alive
    assert self._parent_pid == os.getpid(), 'can only test a child process'
```

```
AssertionError: can only test a child process
Exception ignored in: <function _MultiProcessingDataLoaderIter.__del__ at
0x7f847b973790>
Traceback (most recent call last):
  File "/usr/local/lib/python3.8/dist-packages/torch/utils/data/dataloader.py",
line 1466, in __del__
    self._shutdown_workers()
  File "/usr/local/lib/python3.8/dist-packages/torch/utils/data/dataloader.py",
line 1449, in _shutdown_workers
    if w.is_alive():
  File "/usr/lib/python3.8/multiprocessing/process.py", line 160, in is_alive
    assert self._parent_pid == os.getpid(), 'can only test a child process'
AssertionError: can only test a child process
```

```
epoch 1 ; train loss 1.2652712787224498
HR for this epoch: 0.11189207572837546
NDCG for this epoch: 0.05129219891600043
epoch 2 ; train loss 0.2891841279305167
HR for this epoch: 0.1037864187733817
NDCG for this epoch: 0.04698228078028382
epoch 3 ; train loss 0.2908086077067391
HR for this epoch: 0.11965763518875411
NDCG for this epoch: 0.05276349436364658
epoch 4 ; train loss 0.8198877680811073
HR for this epoch: 0.1126289536333749
NDCG for this epoch: 0.04961553974106485
epoch 5 ; train loss 0.29014177298856025
HR for this epoch: 0.10389978460492008
NDCG for this epoch: 0.047023734293272765
epoch 6 ; train loss 0.5093885977113659
HR for this epoch: 0.11580319691644939
NDCG for this epoch: 0.05240116908703522
```

8 b1-matrix factorization model

8.0.1 explicit

[ ]:
```python
def predict_score_of_target_user_and_movie(k, pu, qi, bu, bi, user, movie, avg):
    bi.setdefault(movie, 0)
    bu.setdefault(user, 0)
    pu.setdefault(user, np.random.random((k,1)))
    qi.setdefault(movie, np.random.random((k,1)))
    pu_ = np.array(pu[user], dtype=np.float64)
    qi_ = np.array(qi[movie], dtype=np.float64)
    rating = np.sum(qi_ * pu_) + bi[movie] + bu[user] + avg
    if rating > 5:
```

```
        rating = 5
    if rating < 1:
        rating = 1
    return rating
```

[ ]:
```
def calculate_ndcg(ratings):
    result = 0
    for i in range(1, len(ratings)):
        result += ratings[i][1] / math.log(i+1,2)
    result += ratings[0][1]
    return result
```

[ ]:
```
train_users = list(train_df['user'].unique())
train_movies = list(train_df['movie'].unique())
```

[ ]:
```
def train_and_evaluate(train_df, test_df, user_movie_train, user_movie_test,␣
↪steps, gamma, Lambda, k, pu, qi, bu, bi,avg):
    train_users = list(train_df['user'].unique())
    train_movies = list(train_df['movie'].unique())
    for user in train_users:
        pu.setdefault(user, np. random.random((k,1))/10*np.sqrt(k))
        bu.setdefault(user,0)
    for movie in train_movies:
        qi.setdefault(movie,np.random.random((k,1))/10*np.sqrt(k))
        bi.setdefault(movie,0)
    for step in range(steps):
        print('step',step)
        random.shuffle(train_users)
        rmse_sum, mape = 0, 0
        count = 0
        for iter, row in train_df.iterrows():
            count += 1

            movie, user, true_rating = row[0],row[1],row[2]
            pred_rating = predict_score_of_target_user_and_movie(k, pu, qi, bu,␣
↪bi, user, movie, avg)
            eui = true_rating - pred_rating
            rmse_sum += eui ** 2
            mape += abs(eui) / float(true_rating)
            #print(target_user, target_movie, eui)
            qi[movie] += gamma * (eui * pu[user] - Lambda * qi[movie])
            pu[user] += gamma * (eui * qi[movie] - Lambda * pu[user])
            bu[user] += gamma * (eui - Lambda * bu[user])
            bi[movie] += gamma * (eui - Lambda * bi[movie])
        gamma = gamma * 0.95
        rmse = np.sqrt(rmse_sum/len(train_df))
        mape = mape / len(train_df) * 100
```

```
        print(f"the rmse for this step on training data is {rmse},"
          f"the mape for this step on training data is {mape}")
        rmse_sum, mape = 0, 0
        #evaluation on test dataset
        for iter, row in test_df.iterrows():
            movie, user,true_rating = row[0], row[1], row[2]
            pred_rating = predict_score_of_target_user_and_movie(k, pu, qi, bu,␣
 ↪bi, user, movie, avg)
            eui = float(true_rating) - pred_rating
            rmse_sum += eui ** 2
            mape += abs(eui) / float(true_rating)
        rmse = np.sqrt(rmse_sum/len(test_df))
        mape = mape/len(test_df)*100
        print(f"the rmse for this step on test data is {rmse},"
          f"the mape for this step on test data is {mape}")
k = 32
steps = 20
gamma = 0.05
Lambda = 0.15
pu = {}
qi = {}
bu = {}
bi = {}
train_df = train_df.iloc[1:]
train_df['rating']=pd.to_numeric(train_df['rating'])
avg = np.mean(train_df['rating'])
print(avg)
train_and_evaluate(train_df, test_df, user_movie_train, user_movie_test, steps,␣
 ↪gamma, Lambda, k, pu, qi, bu, bi,avg)
```

[ ]:

8.0.2 implicit

[ ]:
```
def tanh(x):
    t=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
    return t
```

[ ]:
```
def predict_score_of_target_user_and_movie(k, pu, qi, bu, bi, user, movie):
    bi.setdefault(movie, 0)
    bu.setdefault(user, 0)
    pu.setdefault(user, np.random.random((k,1))/10)
    qi.setdefault(movie, np.random.random((k,1))/10)
    pu_ = np.array(pu[user], dtype=np.float64)
    qi_ = np.array(qi[movie], dtype=np.float64)
```

```
    rating = np.sum(qi_ * pu_) + bi[movie] + bu[user]
    return tanh(rating)
```

[ ]:
```
#evaluation part
def hit(ng_item, pred_items):
    if ng_item in pred_items:
        return 1
    return 0

def ndcg(ng_item, pred_items):
    if ng_item in pred_items:
        index = pred_items.index(ng_item)
        return np.reciprocal(np.log2(index+2))
    return 0
```

[ ]:
```
def train_and_evaluate(train_df_implicit, test_df_implicit, user_movie_train,␣
↪user_movie_test, steps, gamma, Lambda, k, pu, qi, bu, bi,avg):
    train_users = list(train_df_implicit['user'].unique())
    train_movies = list(train_df_implicit['movie'].unique())
    for user in train_users:
        pu.setdefault(user, np. random.random((k,1))/10*np.sqrt(k))
        bu.setdefault(user,0)
    for movie in train_movies:
        qi.setdefault(movie,np.random.random((k,1))/10*np.sqrt(k))
        bi.setdefault(movie,0)
    for step in range(steps):
        print('step',step)
        random.shuffle(train_users)
        count = 0
        for iter, row in train_df_implicit.iterrows():

            user,movie, true_rating = row[1],row[0],row[2]
            pred_rating = predict_score_of_target_user_and_movie(k, pu, qi, bu,␣
↪bi, user, movie)
            eui = true_rating - pred_rating
            #print(target_user, target_movie, eui)
            qi[movie] += gamma * (eui * pu[user] - Lambda * qi[movie])
            pu[user] += gamma * (eui * qi[movie] - Lambda * pu[user])
            bu[user] += gamma * (eui - Lambda * bu[user])
            bi[movie] += gamma * (eui - Lambda * bi[movie])
        gamma = gamma * 0.95
        #evaluation on test dataset
        HR, NDCG = [],[]
        for i in range(0,len(test_df_implicit)-101,505):
            ratings= []
            for j in range(i,i+101):
                row = test_df_implicit.iloc[j]
```

```
                user,movie,true_rating = row[0], row[1], row[2]
                pred_rating = predict_score_of_target_user_and_movie(k, pu, qi,␣
 ↪bu, bi, user, movie)
                ratings.append([movie,pred_rating])
            ng_item = test_df_implicit.iloc[i]['movie']
            recommends=list(sorted(ratings,key=lambda x: x[1],reverse=True))[:
 ↪10]
            recommends = [item[0] for item in recommends]
            HR.append(hit(ng_item, recommends))
            NDCG.append(ndcg(ng_item, recommends))
        HR = np.mean(HR)

        NDCG =np.mean(NDCG)
        print(f"the HR for this step on test data is {HR},"
          f"the NDCG for this step on test data is {NDCG}")
k = 32
steps = 20
gamma = 0.001
Lambda = 0.15
pu = {}
qi = {}
bu = {}
bi = {}
train_df = train_df.iloc[1:]
train_df['rating']=pd.to_numeric(train_df['rating'])
train_and_evaluate(train_df_implicit, test_df_implicit, user_movie_train,␣
 ↪user_movie_test, steps, gamma, Lambda, k, pu, qi, bu, bi,avg)
```

[ ]:

9 fushion of GMF and MLP

[ ]:
```
def calculate_aout(x):
    return 1/(1+np.exp(-x))
```

[ ]:
```
def calculate_ndcg(ratings):
    result = 0
    for i in range(1, len(ratings)):
        result += ratings[i][1] / math.log(i+1,2)
    result += ratings[0][1]
    return result
```

[ ]:
```
class GMF:
    def __init__(self):
        self.pu = {}
```

```
        self.qi = {}
        self.h = {}

GMF_model = GMF()
```

[ ]:
```
class MLP:
    def __init__(self):
        self.pu = {}
        self.qi = {}
        self.w_hidden1 = {}
        self.w_hidden2 = {}
        self.b_hidden1 = {}
        self.b_hidden2 = {}
        self.w_output = {}
        self.b_output = {}

MLP_model = MLP()
```

[ ]:
```
def predict_score_of_target_user_and_movie(k,GMF_model,MLP_model , user, movie):
    #MLP
    MLP_model.pu.setdefault(user, np.random.random((k,1)))
    MLP_model.qi.setdefault(movie, np.random.random((k,1)))
    MLP_model.w_hidden1.setdefault(user, np.random.random((2*k,50)))
    MLP_model.w_hidden2.setdefault(user, np.random.random((50,25)))
    MLP_model.w_output.setdefault(user,np.random.random((25,1)))
    MLP_model.b_hidden1.setdefault(user, np.random.random((50,1)))
    MLP_model.b_hidden2.setdefault(user, np.random.random((25,1)))
    MLP_model.b_output.setdefault(user,0)
    Input = np.concatenate((MLP_model.pu[user],MLP_model.qi[movie]))
    output_1 = calculate_aout(np.dot(np.transpose(MLP_model.
↪w_hidden1[user]),Input)/np.sum(Input)+MLP_model.b_hidden1[user])
    output_2 = calculate_aout(np.dot(np.transpose(MLP_model.
↪w_hidden2[user]),output_1)/np.sum(output_1)+MLP_model.b_hidden2[user])
    pred_rating_MLP = float(np.dot(np.transpose(MLP_model.
↪w_output[user]),output_2)/np.sum(output_2))+MLP_model.b_output[user]
    #GMF
    GMF_model.pu.setdefault(user, np.random.random((k,1)))
    GMF_model.qi.setdefault(movie, np.random.random((k,1)))
    GMF_model.h.setdefault(user, np.random.random((k,1)))
    pu_ = np.array(GMF_model.pu[user], dtype=np.float64)
    qi_ = np.array(GMF_model.qi[movie], dtype=np.float64)
    pred_rating_GMF =float(np.dot(np.transpose(GMF_model.h[user]),(qi_ * pu_)))
    rating = calculate_aout(pred_rating_MLP + pred_rating_GMF)
    #= calculate_aout(float(np.dot(np.transpose(MLP.w_output[user]),output_2)/
↪np.sum(output_2))+MLP.b_output[user
    return Input,output_1,output_2,rating
```

```
[ ]: def␣
↪train(train_users,train_movies,user_movie_train,user_movie_validation,steps,gamma,Lambda,k,
↪MLP_model, output_path):
        result = {}
        for user in train_users:
            GMF_model.pu.setdefault(user, np.random.random((k,1))/10*np.sqrt(k))
            GMF_model.h.setdefault(user, np.random.random((k,1))/10*np.sqrt(k))
            MLP_model.pu.setdefault(user, np.random.random((k,1))/10*np.sqrt(k))
            MLP_model.w_hidden1.setdefault(user,np.random.random((2*k,50))/
↪10*np.sqrt(k))
            MLP_model.w_hidden2.setdefault(user, np.random.random((50,25))/
↪10*np.sqrt(50))
            MLP_model.w_output.setdefault(user,np.random.random((25,1))/10*np.
↪sqrt(25))
            MLP_model.b_hidden1.setdefault(user, np.random.random((50,1))/10*np.
↪sqrt(50))
            MLP_model.b_hidden2.setdefault(user, np.random.random((25,1))/10*np.
↪sqrt(25))
            MLP_model.b_output.setdefault(user,0)
        for movie in train_movies:
            MLP_model.qi.setdefault(movie, np.random.random((k,1))/10*np.
↪sqrt(k))
            GMF_model.qi.setdefault(movie, np.random.random((k,1))/10*np.
↪sqrt(k))
        for step in range(steps):
            print('step',step)
            HR_validation = 0
            NDCG_validation = 0
            count = 0
            for user in train_users:
                count += 1
                if count % 10000 == 0:
                    print('user count', count)
                    print(HR_validation/count)
                    print(NDCG_validation/count)

                ratings = []
                hr_validation = 0
                movies = eval(user_movie_train[user]['movie'])
                random.shuffle(movies)
                for movie, true_rating in movies:
                    Input,output_1,output_2, pred_rating =␣
↪predict_score_of_target_user_and_movie(k,GMF_model,MLP_model , user, movie)
                    ratings.append([pred_rating, true_rating])
```

```
                ratings_pred = sorted(ratings, key=lambda item: item[0],␣
↪reverse = True)[:10]

                for i in range(len(ratings_pred)):
                    Input,output_1,output_2, pred_rating =␣
↪predict_score_of_target_user_and_movie(k,GMF_model,MLP_model , user, movie)
                    eui = float(ratings_pred[i][1]) - pred_rating
                    w_output_add =␣
↪gamma*eui*pred_rating*(1-pred_rating)*output_2
                    MLP_model.b_output[user] +=␣
↪gamma*eui*pred_rating*(1-pred_rating)
                    w_hidden2_add = np.transpose(np.
↪dot(gamma*eui*pred_rating*(1-pred_rating)*MLP_model.
↪w_output[user]*output_2*(1-output_2),np.transpose(output_1)))
                    MLP_model.b_hidden2[user] +=␣
↪gamma*eui*pred_rating*(1-pred_rating)*MLP_model.
↪w_output[user]*output_2*(1-output_2)
                    MLP_model.w_hidden1[user] += np.transpose(np.dot(np.
↪dot(MLP_model.
↪w_hidden2[user],gamma*eui*pred_rating*(1-pred_rating)*MLP_model.
↪w_output[user]*output_2*(1-output_2))*output_1*(1-output_1),np.
↪transpose(Input)))
                    #w_hidden1[user] +=␣
↪2*eui*pred_rating*(1-pred_rating)*w_output[user]*output_2*(1-output_2)*w_hidden2[user]*outp
                    MLP_model.b_hidden1[user]+=np.dot(MLP_model.
↪w_hidden2[user],gamma*eui*pred_rating*(1-pred_rating)*MLP_model.
↪w_output[user]*output_2*(1-output_2))*output_1*(1-output_1)
                    #b_hidden1[user] +=␣
↪2*eui*pred_rating*(1-pred_rating)*w_output[user]*output_2*(1-output_2)*w_hidden2[user]*outp
                    MLP_model.w_hidden2[user] += w_hidden2_add
                    MLP_model.w_output[user] += w_output_add
                    MLP_model.qi[movie] += gamma *(eui * MLP_model.
↪pu[user]-Lambda * MLP_model.qi[movie])
                    MLP_model.pu[user] += gamma * (eui * MLP_model.qi[movie] -␣
↪Lambda * MLP_model.pu[user])

                    GMF_model.h[user] += gamma * eui * pred_rating *␣
↪(1-pred_rating) * np.sum(GMF_model.qi[movie]*GMF_model.pu[user])
                    GMF_model.qi[movie] += gamma *(eui * GMF_model.
↪pu[user]-Lambda * GMF_model.qi[movie])
                    GMF_model.pu[user] += gamma * (eui * GMF_model.qi[movie] -␣
↪Lambda * GMF_model.pu[user])

        return GMF_model, MLP_model
k = 32
steps = 3
```

```
gamma = 0.05
Lambda = 0.15
pu = {}
qi = {}
w_hidden1 = {}
w_hidden2 = {}
b_hidden1 = {}
b_hidden2 = {}
w_output = {}
b_output = {}
#print(predict_target_movie_score_from_target_user(k, pu, qi, target_user,␣
↪target_movie))
#pu, qi, bu, bi = train(movie_user, user_movie, steps, gamma ,Lambda, k ,pu,␣
↪qi, bu, bi, avg)
output_path = 'n3_train_32.csv'
GMF_model, MLP_model =␣
↪train(train_users,train_movies,user_movie_train,user_movie_validation,steps,gamma,Lambda,k,
↪MLP_model, output_path)
```

[ ]:
```
def test(test_users,test_movies,user_movie_test,k,GMF_model, MLP_model,␣
↪output_path):
    for user in train_users:
        GMF_model.pu.setdefault(user, np.random.random((k,1))/10*np.sqrt(k))
        GMF_model.h.setdefault(user, np.random.random((k,1))/10*np.sqrt(k))
        MLP_model.pu.setdefault(user, np.random.random((k,1))/10*np.sqrt(k))
        MLP_model.w_hidden1.setdefault(user,np.random.random((2*k,50))/10*np.
↪sqrt(k))
        MLP_model.w_hidden2.setdefault(user, np.random.random((50,25))/10*np.
↪sqrt(50))
        MLP_model.w_output.setdefault(user,np.random.random((25,1))/10*np.
↪sqrt(25))
        MLP_model.b_hidden1.setdefault(user, np.random.random((50,1))/10*np.
↪sqrt(50))
        MLP_model.b_hidden2.setdefault(user, np.random.random((25,1))/10*np.
↪sqrt(25))
        MLP_model.b_output.setdefault(user,0)
    for movie in train_movies:
        MLP_model.qi.setdefault(movie, np.random.random((k,1))/10*np.sqrt(k))
        GMF_model.qi.setdefault(movie, np.random.random((k,1))/10*np.sqrt(k))
    HR_test = 0
    NDCG_test = 0
    count = 0
    for user in test_users:
        count += 1
        if count % 10000 == 0:
            print('user count', count)
```

```
            print(HR_test/count)
            print(NDCG_test/count)

        ratings = []
        hr_test = 0
        print(user_movie_test[user])
        movies = eval(user_movie_test[user]['movie'])
        random.shuffle(movies)
        for movie, true_rating in movies:
            Input,output_1,output_2, pred_rating =␣
↪predict_score_of_target_user_and_movie(k,GMF_model,MLP_model , user, movie)
            ratings.append([pred_rating, true_rating])


        ratings_pred = sorted(ratings, key=lambda item: item[0], reverse =␣
↪True)[:10]
        ratings_true = sorted(ratings, key=lambda item: item[1], reverse =␣
↪True)[:10]

        dcg = calculate_ndcg(ratings_pred)
        idcg = 1
        if idcg != 0:
            NDCG_test += dcg/idcg
        for i in range(len(ratings_pred)):
            if ratings_pred[i][1] == 1:
                hr_test = 1
                break


        HR_test += hr_test
    HR_test = HR_test/len(test_users)
    NDCG_test = NDCG_test/len(test_users)

    print(f"the value of HR(10) on test data is {HR_test},"
        f"the value of NDCG(10) on test data is {NDCG_test}")
    output_path = 'n3_test_16.csv'
    result = {'HR':[HR_test], 'NDCG':[NDCG_test]}
    pd.DataFrame(result).to_csv(output_path)
test(test_users,test_movies,user_movie_test,k,GMF_model, MLP_model, output_path)
```

[2]:
```
!jupyter nbconvert --to PDF "final_project.ipynb"
```

```
[NbConvertApp] WARNING | pattern 'final_project.ipynb' matched no files
This application is used to convert notebook files (*.ipynb)
        to various other formats.

        WARNING: THE COMMANDLINE INTERFACE MAY CHANGE IN FUTURE RELEASES.
```

```
Options
=======
The options below are convenience aliases to configurable class-options,
as listed in the "Equivalent to" description-line of the aliases.
To see all configurable class-options for some <cmd>, use:
    <cmd> --help-all

--debug
    set log level to logging.DEBUG (maximize logging output)
    Equivalent to: [--Application.log_level=10]
--show-config
    Show the application's configuration (human-readable format)
    Equivalent to: [--Application.show_config=True]
--show-config-json
    Show the application's configuration (json format)
    Equivalent to: [--Application.show_config_json=True]
--generate-config
    generate default config file
    Equivalent to: [--JupyterApp.generate_config=True]
-y
    Answer yes to any questions instead of prompting.
    Equivalent to: [--JupyterApp.answer_yes=True]
--execute
    Execute the notebook prior to export.
    Equivalent to: [--ExecutePreprocessor.enabled=True]
--allow-errors
    Continue notebook execution even if one of the cells throws an error and
include the error message in the cell output (the default behaviour is to abort
conversion). This flag is only relevant if '--execute' was specified, too.
    Equivalent to: [--ExecutePreprocessor.allow_errors=True]
--stdin
    read a single notebook file from stdin. Write the resulting notebook with
default basename 'notebook.*'
    Equivalent to: [--NbConvertApp.from_stdin=True]
--stdout
    Write notebook output to stdout instead of files.
    Equivalent to: [--NbConvertApp.writer_class=StdoutWriter]
--inplace
    Run nbconvert in place, overwriting the existing notebook (only
            relevant when converting to notebook format)
    Equivalent to: [--NbConvertApp.use_output_suffix=False
--NbConvertApp.export_format=notebook --FilesWriter.build_directory=]
--clear-output
    Clear output of current file and save in place,
            overwriting the existing notebook.
    Equivalent to: [--NbConvertApp.use_output_suffix=False
--NbConvertApp.export_format=notebook --FilesWriter.build_directory=
```

```
--ClearOutputPreprocessor.enabled=True]
--no-prompt
    Exclude input and output prompts from converted document.
    Equivalent to: [--TemplateExporter.exclude_input_prompt=True
--TemplateExporter.exclude_output_prompt=True]
--no-input
    Exclude input cells and output prompts from converted document.
            This mode is ideal for generating code-free reports.
    Equivalent to: [--TemplateExporter.exclude_output_prompt=True
--TemplateExporter.exclude_input=True]
--log-level=<Enum>
    Set the log level by value or name.
    Choices: any of [0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR',
'CRITICAL']
    Default: 30
    Equivalent to: [--Application.log_level]
--config=<Unicode>
    Full path of a config file.
    Default: ''
    Equivalent to: [--JupyterApp.config_file]
--to=<Unicode>
    The export format to be used, either one of the built-in formats
            ['asciidoc', 'custom', 'html', 'latex', 'markdown', 'notebook',
'pdf', 'python', 'rst', 'script', 'slides']
            or a dotted object name that represents the import path for an
            `Exporter` class
    Default: 'html'
    Equivalent to: [--NbConvertApp.export_format]
--template=<Unicode>
    Name of the template file to use
    Default: ''
    Equivalent to: [--TemplateExporter.template_file]
--writer=<DottedObjectName>
    Writer class used to write the
                                        results of the conversion
    Default: 'FilesWriter'
    Equivalent to: [--NbConvertApp.writer_class]
--post=<DottedOrNone>
    PostProcessor class used to write the
                                        results of the conversion
    Default: ''
    Equivalent to: [--NbConvertApp.postprocessor_class]
--output=<Unicode>
    overwrite base name use for output files.
                can only be used when converting one notebook at a time.
    Default: ''
    Equivalent to: [--NbConvertApp.output_base]
--output-dir=<Unicode>
```

```
    Directory to write output(s) to. Defaults
                                    to output to the directory of each notebook.
To recover
                                    previous default behaviour (outputting to the
current
                                    working directory) use . as the flag value.
    Default: ''
    Equivalent to: [--FilesWriter.build_directory]
--reveal-prefix=<Unicode>
    The URL prefix for reveal.js (version 3.x).
            This defaults to the reveal CDN, but can be any url pointing to a
copy
            of reveal.js.
            For speaker notes to work, this must be a relative path to a local
            copy of reveal.js: e.g., "reveal.js".
            If a relative path is given, it must be a subdirectory of the
            current directory (from which the server is run).
            See the usage documentation
            (https://nbconvert.readthedocs.io/en/latest/usage.html#reveal-js-
html-slideshow)
            for more details.
    Default: ''
    Equivalent to: [--SlidesExporter.reveal_url_prefix]
--nbformat=<Enum>
    The nbformat version to write.
            Use this to downgrade notebooks.
    Choices: any of [1, 2, 3, 4]
    Default: 4
    Equivalent to: [--NotebookExporter.nbformat_version]

Examples
--------

    The simplest way to use nbconvert is

            > jupyter nbconvert mynotebook.ipynb

            which will convert mynotebook.ipynb to the default format (probably
HTML).

            You can specify the export format with `--to`.
            Options include ['asciidoc', 'custom', 'html', 'latex', 'markdown',
'notebook', 'pdf', 'python', 'rst', 'script', 'slides'].

            > jupyter nbconvert --to latex mynotebook.ipynb

            Both HTML and LaTeX support multiple output templates. LaTeX
includes
```

```
            'base', 'article' and 'report'.  HTML includes 'basic' and 'full'.
You
            can specify the flavor of the format used.

            > jupyter nbconvert --to html --template basic mynotebook.ipynb

            You can also pipe the output to stdout, rather than a file

            > jupyter nbconvert mynotebook.ipynb --stdout

            PDF is generated via latex

            > jupyter nbconvert mynotebook.ipynb --to pdf

            You can get (and serve) a Reveal.js-powered slideshow

            > jupyter nbconvert myslides.ipynb --to slides --post serve

            Multiple notebooks can be given at the command line in a couple of
            different ways:

            > jupyter nbconvert notebook*.ipynb
            > jupyter nbconvert notebook1.ipynb notebook2.ipynb

            or you can specify the notebooks list in a config file, containing::

                c.NbConvertApp.notebooks = ["my_notebook.ipynb"]

            > jupyter nbconvert --config mycfg.py

To see all available configurables, use `--help-all`.
```

[ ]: