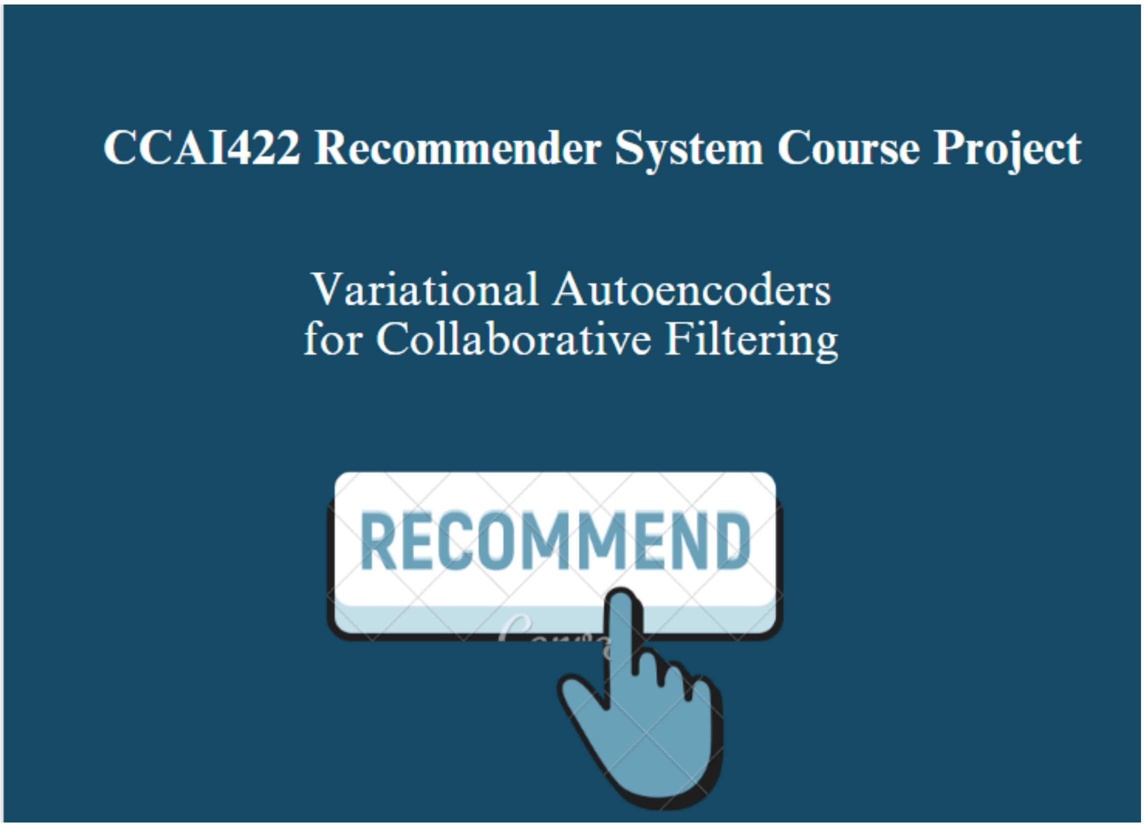
Variational Autoencoders for Collaborative Filtering Implementaion



Team Contributions:

- **Bedoor Ayad,2005061**
- **Raneem Alomari 2006352,**
- **Deema Al-saygh.2006085**

Introduction:

In our journey through a recommendation system course project, we embarked on an exciting exploration of recommendation system and dive deep into its concepts, implement its findings, and push the boundaries of its capabilities to improve and innovate in the field of personalized recommendations. The major challenging "small-data" problem where most users only interact with a tiny proportion of the items and our goal is to collectively make informed inference about each user's preference . we find that to make use of the sparse signals from users and avoid overfitting, we build a probabilistic latent-variable model that shares statistical strength among users and items. Empirically, we show that employing a principled Bayesian approach is more robust regardless of the scarcity of the data. However, we did not stop at mere replication. We fine-tuned parameters of the Variational Autoencoders (VAEs) , explored alternative dataset to create a functional system that could generate personalized recommendations.

This notebook is base on the [Variational Autoencoders for Collaborative Filtering paper] (https://arxiv.org/pdf/1802.05814v1.pdf)

Library

```
!pip install imbalanced-learn
```

```
Requirement already satisfied: imbalanced-learn in /opt/conda/li
Requirement already satisfied: joblib>=0.11 in /opt/conda/lib/py
Requirement already satisfied: numpy>=1.13.3 in /opt/conda/lib/p
WARNING: Retrying (Retry(total=4, connect=None, read=None, redir
WARNING: Retrying (Retry(total=3, connect=None, read=None, redir
WARNING: Retrying (Retry(total=2, connect=None, read=None, redir
WARNING: Retrying (Retry(total=1, connect=None, read=None, redir
WARNING: Retrying (Retry(total=0, connect=None, read=None, redir
WARNING: Retrying (Retry(total=4, connect=None, read=None, redir
WARNING: Retrying (Retry(total=3, connect=None, read=None, redir
WARNING: Retrying (Retry(total=2, connect=None, read=None, redir
WARNING: Retrying (Retry(total=1, connect=None, read=None, redir
WARNING: Retrying (Retry(total=0, connect=None, read=None, redir
ERROR: Could not find a version that satisfies the requirement s
ERROR: No matching distribution found for scikit-learn>=0.24
```

```
# Imports the shutil module, which provides high-level file operat
import shutil

# Imports the sys module, which provides access to some variables
#used or maintained by the interpreter and functions that interact
import sys
```

```
# Imports the NumPy library and assigns it the alias 'np' for ease
import numpy as np
# Imports the 'sparse' module from SciPy, which deals with sparse
from scipy import sparse

# Imports the pyplot module from Matplotlib and assigns it the ali
import matplotlib.pyplot as plt

# Magic command in Jupyter Notebook to display Matplotlib plots ir
#%matplotlib inline

# Imports the Seaborn library for statistical data visualization c
import seaborn as sn
# Sets the default Seaborn theme and applies aesthetic preferences
sn.set()

# Imports the Pandas library for data manipulation and analysis ar
import pandas as pd

# Imports the random module, providing functions for generating rc
import random
# Imports TensorFlow, a popular deep learning framework, and assig
import tensorflow as tf

# Imports the bottleneck library, which provides specialized array
import bottleneck as bn

# Imports the os module, which provides a way of using operating s
import os
```

Data Loading

```
#Loading data
raw_data1 = pd.read_csv("/kaggle/input/netflix-movie-rating-datase
```

Data visualization :

```
# Displays the first few rows of the DataFrame 'raw_data', allowir
#structure and contents.
raw_data1.head()
```

```
   User_ID  Rating  Movie_ID
0   712664       5         3
1  1331154       4         3
2  2632461       3         3
3    44937       5         3
4   656399       4         3
```

	User_ID	Rating	Movie_ID
0	712664	5	3
1	1331154	4	3
2	2632461	3	3
3	44937	5	3
4	656399	4	3

Data Preprocessing:

```
raw_data1.Rating.nunique() # values are integer {5 , 4 , 3, 2, 1}
```

```
5
```

Binarizing the ratings

There are some database of ratings ranging from 0.5 to 5. so we check if there float we Binarizing data if not then no need for this step. Binarizing the ratings can be useful in certain scenarios, especially when dealing with recommendation systems

```
# Check if the 'Rating' column contains float values
```

```
if raw_data1['Rating'].dtype == 'float64':
    # Binarize the ratings (assuming a threshold, e.g., 3.5)
    threshold = 3.5
    raw_data1['Binarized_Rating'] = (raw_data1['Rating'] >= thresh
    # Drop the original 'Rating' column if needed
    raw_data1 = raw_data1.drop('Rating', axis=1)
    # Now you have a new column 'Binarized_Rating' with 1 for rati

else:
    # No need to binarize if the ratings are not float
    print("The 'Rating' column is not of float type.")
```

```
The 'Rating' column is not of float type.
```

Handeling imbalanced data using Undersampling techniques

```
# Counts the number of ratings equal to 5 in the 'Rating' column c
print("# Ratings = 5: ",sum(raw_data1['Rating']==5))

# Counts the number of ratings equal to 4 in the 'Rating' column c
print("# Ratings = 4: ",sum(raw_data1['Rating']==4))

# Counts the number of ratings equal to 3 in the 'Rating' column c
print("# Ratings = 3: ",sum(raw_data1['Rating']==3))

# Counts the number of ratings equal to 2 in the 'Rating' column c
print("# Ratings = 2: ",sum(raw_data1['Rating']==2))

# Counts the number of ratings equal to 1 in the 'Rating' column c
print("# Ratings = 1: ",sum(raw_data1['Rating']==1))

# Displays the first few rows of the 'raw_data' DataFrame
raw_data1.head()
```

```
# Ratings = 5:  3769803
# Ratings = 4:  5901368
# Ratings = 3:  5185650
# Ratings = 2:  1759281
# Ratings = 1:  721356
```

```
   User_ID  Rating  Movie_ID
```

```
0   712664       5         3
1  1331154       4         3
2  2632461       3         3
3    44937       5         3
4   656399       4         3
```

	User_ID	Rating	Movie_ID
0	712664	5	3
1	1331154	4	3
2	2632461	3	3
3	44937	5	3
4	656399	4	3

It seems like the distribution is not perfectly balanced, but it's common to have some class imbalance in real-world datasets. In Our case, Ratings 4 and 3 have more instances compared to others.

Handling class imbalance is crucial in machine learning, especially when certain classes have significantly fewer instances than others. Here techniques we considered: Undersampling: Decrease the number of instances in the majority class by randomly removing examples.

```
from imblearn.under_sampling import RandomUnderSampler
import pandas as pd

# Assuming raw_data is your DataFrame with 'Rating' as the target
X = raw_data1.drop('Rating', axis=1)
y = raw_data1['Rating']

# Calculate the target number of instances for each class
target_instances = min(raw_data1['Rating'].value_counts())

# Set the target ratio for each class
undersample_ratio = {1: target_instances, 2: target_instances, 3:

# Apply random undersampling
```

```python
undersampler = RandomUnderSampler(sampling_strategy=undersample_ra
X_resampled, y_resampled = undersampler.fit_resample(X, y)

# Now X_resampled and y_resampled contain the undersampled dataset
```

```python
# Concatenate the undersampled data (X_resampled, y_resampled) wit
raw_data = pd.concat([X_resampled, pd.Series(y_resampled, name='Ra

# Now raw_data is updated with the undersampled data
raw_data.head()
```

```
   User_ID  Movie_ID  Rating
0  2330911      1582       1
1   446767      3220       1
2  2502462      1138       1
3  2402461       483       1
4   122051      2783       1
```

	User_ID	Movie_ID	Rating
0	2330911	1582	1
1	446767	3220	1
2	2502462	1138	1
3	2402461	483	1
4	122051	2783	1

```python
# Counts the number of ratings equal to 5 in the 'Rating' column c
print("# Ratings = 5: ",sum(raw_data['Rating']==5))

# Counts the number of ratings equal to 4 in the 'Rating' column c
print("# Ratings = 4: ",sum(raw_data['Rating']==4))

# Counts the number of ratings equal to 3 in the 'Rating' column c
print("# Ratings = 3: ",sum(raw_data['Rating']==3))

# Counts the number of ratings equal to 2 in the 'Rating' column c
print("# Ratings = 2: ",sum(raw_data['Rating']==2))

# Counts the number of ratings equal to 1 in the 'Rating' column c
print("# Ratings = 1: ",sum(raw_data['Rating']==1))
```

```
# Ratings = 5:  721356
```

```
# Ratings = 4:  721356
# Ratings = 3:  721356
# Ratings = 2:  721356
# Ratings = 1:  721356
```

now each class has same number of instances

Data Splitting Procedure

- Select 10K users as excluded users, 10K users as validation users, and the remaining users for training.
- Use all items from the training users as the item set.
- For each validation and test user, subsample 80% as fold data and the remainder for prediction.

```
# Function that counts the number of specific items grouped
# Display the count of occurrences for each unique 'Movie_ID' in t
print(raw_data[['Movie_ID']].groupby('Movie_ID', as_index=False).s

# Define a function 'get_count'that takes two parameters: 'data' (
#and 'item' (specific item to count).
def get_count( data, item ):

# Group the data by the specified 'item' and count the occurrences
    playcount_groupbyid = data[[item]].groupby(item, as_index=Fals

# Calculate the count of occurrences for each group and store it i
    count = playcount_groupbyid.size()['size']

# Return the count of occurrences for the specified 'item'.
    return count
```

```
<bound method GroupBy.size of <pandas.core.groupby.generic.DataF
```

```
# Calls the 'get_count' function to count the occurrences of uniqu
#in the 'raw_data' DataFrame.
get_count(raw_data, 'User_ID')
```

```
0          28
```

```
1         27
2         48
3         32
4         33
          ..
143449     9
143450    10
143451    17
143452     8
143453     9
Name: size, Length: 143454, dtype: int64
```

Filter triples (data, user, item)

```
#The function of filtered triplets (data, user count, and item cou
def filter_triplets( data, min_uc = 5, min_sc = 5 ):
    # Considering triplets only for items selected by at least mir
    if min_sc > 0:

        # Counting the occurrences of each Movie_ID and filtering
        item_count = get_count( data, 'Movie_ID')

        # Filtering data by Movie_IDs meeting the minimum count re
        data = data[ data[ 'Movie_ID' ].isin(item_count.index[ ite

     # Considering triplets for users who interacted with at least
    if min_uc > 0:

        # Counting the occurrences of each User_ID and filtering c
        user_count = get_count( data, 'User_ID' )

        # Filtering data by User_IDs meeting the minimum count rec
        data = data[ data[ 'User_ID' ].isin(user_count.index[ user

    # Updating the item and user count after the filters have beer
    item_count = get_count( data, 'Movie_ID' )# Recalculating iter
    user_count = get_count( data, 'User_ID' )# Recalculating user
    return data, user_count, item_count
```

Considering items that were rated by at least 5 users and users who rated at least 5 movies.

```
# Filtering the raw_data DataFrame based on default minimum user c
#using the filter_triplets function.
# The resulting filtered data, user_activity, and item_popularity
#and assigned to the respective variables.
raw_data, user_activity, item_popularity = filter_triplets(raw_dat
```

spread metric

The numerator of the spread metric is calculated by counting the total number of ratings contained in the ratings matrix. The denominator of the spread metric is calculated by multiplying the number of users by the number of movies in the ratings matrix. The spread is calculated and printed by dividing the numerator by the denominator, subtracting 1, and multiplying by 100. Adding 1.0 ensures that the spread is returned as a decimal rather than an integer.

```
# Calculate the sparsity of the dataset after applying filters
sparsity = 1. * raw_data.shape[0] / (user_activity.shape[0] * item
print("After the filters, there are %d events from %d users and %d
(raw_data.shape[0], user_activity.shape[0], item_popularity.shape[
```

```
After the filters, there are 32748 events from 4608 users and 42
```

```
# Display the shape (dimensions) of the 'raw_data' DataFrame.
print(raw_data.shape)

# Display the shape (dimensions) of the 'item_popularity' DataFram
print(item_popularity.shape)

# Display the shape (dimensions) of the 'user_activity' DataFrame.
print(user_activity.shape)
```

```
(32748, 3)
(421,)
(4608,)
```

```
# Unique index of users
# Extracting unique user IDs as the index from the 'user_activity'
unique_uid = user_activity.index
```

```
# Setting a seed for reproducibility before shuffling the indices
np.random.seed(98765)

# Creating a random permutation of the indices' size to shuffle th
idx_perm = np.random.permutation(unique_uid.size)

# Reordering the unique user IDs based on the permutation
unique_uid = unique_uid[idx_perm]

# Printing the size (number of elements) of the unique user ID arr
print(unique_uid.size)
```

```
4608
```

Creating train/validation/test sets for users

```
# Calculating the total number of unique users in the dataset
n_users = unique_uid.size


# Setting the number of users to be held out or excluded from the

# n_heldout_users represents the number of users that won't be inc
#in the training/validation/test sets
n_heldout_users = 1000

# Creating training set users by taking all users except the last
tr_users = unique_uid[:(n_users - n_heldout_users * 2)]

# Creating validation set users by taking the users from
#n_users - n_heldout_users * 2 to n_users - n_heldout_users
vd_users = unique_uid[(n_users - n_heldout_users * 2): (n_users -


# Creating test set users by taking the last n_heldout_users users
te_users = unique_uid[(n_users - n_heldout_users):]
```

```
# Selecting all the training users' data from our raw_data

# Filtering raw_data to include only rows where the 'User_ID'
#is present in the tr_users list (training set users)
```

```
train_plays = raw_data.loc[raw_data['User_ID'].isin(tr_users)]
train_plays
```

```
         User_ID  Movie_ID  Rating
3120        3423       720       1
5698        3458      1289       1
6638        3595       749       1
6996        3871       438       1
23306        695         8       1
...          ...       ...     ...
3580789     3786      1180       5
3587513     2385       329       5
3593365     3210        28       5
3598161     3786      1324       5
3601142     1442       937       5

[588 rows x 3 columns]
```

	User_ID	Movie_ID	Rating
3120	3423	720	1
5698	3458	1289	1
6638	3595	749	1
6996	3871	438	1
23306	695	8	1
...	...	...	...
3580789	3786	1180	5
3587513	2385	329	5
3593365	3210	28	5
3598161	3786	1324	5
3601142	1442	937	5

588 rows × 3 columns

```
# Unique movies without repetition
# Extracting unique movie IDs from the 'Movie_ID' column of the tr
unique_sid = pd.unique( train_plays[ 'Movie_ID'] )
unique_sid
```

```
array([ 720, 1289,  749,  438,    8,  528,  457, 1046,  723,  95
```

```
        1267,  199,  341,  197, 1027, 1145,  843,   77, 1245,  93
        1202,  334,  329,  554,  187, 1116,  273,  398,  285, 130
        1100,  672,  175,  331,  494,  788,  353,  964,  416,   3
        1073, 1102,  295,  662,  468,  977,  483,   55,  829,  75
         482,  708, 1110, 1250,  427,  390,  256,  846,  985,  55
         986,  424,  108,  127,  413,  692,  918,  660,  312,  28
         406, 1255,  956,  255,  111,  473,  730,  789,  357,   1
         798,  311, 1180,  989,  564,  763, 1305,  993,  305,  18
        1172,  269, 1096,  818,  643,  213,  550,  674, 1216,  66
        1262,  722,  659,  759,  831,  330, 1295, 1324,  670,  29
         143,  705, 1055,  281,  607,  621, 1050, 1224, 1075,  75
        1201,  896, 1012, 1336,  118,  445, 1066,  886, 1068,   8
         859,  907, 1174, 1058,  686,  548, 1329,  931,  638, 116
         152,  385,  535, 1300,  760,  636,  257, 1022,  524,  46
          78,   52, 1011,  110,  900,   46, 1020,  646,  629, 122
         908,  442,  156,  994,  241,  253,  808,  166,  954,  47
         501,  872,  733, 1208,  405,  561, 1256,  840,   97,  36
        1148, 1291,  275,  940,  569,   28, 1060,  962,  252,  10
         443,  270,   18,  215, 1314,  577,  171,  575,  138,  16
```

```python
# Dictionary for users

# Creating a dictionary 'profile2id' where user IDs are mapped to
profile2id = dict((pid,i) for (i, pid) in enumerate(unique_uid))

# Printing the length (number of items) in the 'profile2id' dictio
print(len(profile2id))

# Dictionary for movies (enumerating the indices)
# Creating a dictionary 'show2id' where movie IDs are mapped to un
show2id   = dict((sid, i) for (i, sid) in enumerate(unique_sid))

# Printing the length (number of items) in the 'show2id' dictionar
print(len(show2id))
```

```
4608
219
```

Writing movie files

```python
# Defining the directory path where the files will be stored
DATA_DIR = ''
pro_dir = os.path.join(DATA_DIR, 'pro_sg')
```

```
# Checking if the directory 'pro_dir' does not exist; if not, crea
if not os.path.exists(pro_dir):
    os.makedirs(pro_dir)

# Writing unique movie IDs to a text file named 'unique_sid.txt'
# Opening a file named 'unique_sid.txt' in write mode within the
with open(os.path.join(pro_dir, 'unique_sid.txt'), 'w') as f:
     # Writing each unique movie ID from the 'unique_sid' array to
    for sid in unique_sid:
        f.write('%s\n' % sid)
```

Function to generate training data and test data

```
# Function to split data into training and test sets based on user
def split_train_test_proportion(data, test_prop=0.2):
    # Grouping data by 'User_ID'
    data_grouped_by_user = data.groupby('User_ID')

    # Initializing lists to hold training and test data
    tr_list, te_list = list(), list()

    # Setting a seed for random number generation
    np.random.seed(98765)

    # Iterating through each group (user) in the grouped data
    for i, (_, group) in enumerate(data_grouped_by_user):
        # Calculating the number of items per user
        n_items_u = len(group)

        # If the user has at least 5 items
        if n_items_u >= 5:
            # Creating a boolean mask for the test items
            idx = np.zeros(n_items_u, dtype='bool')
            idx[np.random.choice(n_items_u, size=int(test_prop * n

             # Adding the items to the training and test lists bas
            tr_list.append(group[np.logical_not(idx)])
            te_list.append(group[idx])
        else:
            # If the user has less than 5 items, add all items to
            tr_list.append(group)

        # Printing progress every 10 users sampled
```

```
        if i % 10 == 0:
            print("%d users sampled" % i)
            sys.stdout.flush()

    # Concatenating the training and test data from the lists
    data_tr = pd.concat(tr_list)
    data_te = pd.concat(te_list)

    return data_tr, data_te
```

```
# Location of users in the test data

# Filtering raw_data to include only rows where the 'User_ID'
# is present in the vd_users list (validation set users)
vad_plays = raw_data.loc[raw_data['User_ID'].isin(vd_users)]

# Further filtering vad_plays to include only rows where the 'Movi
#is present in the unique_sid list (unique movie IDs)
vad_plays = vad_plays.loc[vad_plays['Movie_ID'].isin(unique_sid)]
vad_plays
```

```
         User_ID  Movie_ID  Rating
22362       1333      1267       1
24769       4439       361       1
49841       1427       187       1
138841      4439       964       1
155937      1333       127       1
...          ...       ...     ...
3467266      527       571       5
3486848     3392       270       5
3528253      788       443       5
3558040     3487      1110       5
3602389     1067        30       5

[167 rows x 3 columns]
```

	User_ID	Movie_ID	Rating
22362	1333	1267	1
24769	4439	361	1
49841	1427	187	1
138841	4439	964	1
155937	1333	127	1
...	...	...	...

3467266	527	571	5
3486848	3392	270	5
3528253	788	443	5
3558040	3487	1110	5
3602389	1067	30	5

167 rows × 3 columns

```
# Splitting validation data of movies for validation users into tr
# Using the previously defined function split_train_test_proportic
#DataFrame into training and test sets
vad_plays_tr, vad_plays_te = split_train_test_proportion(vad_plays
```

```
0 users sampled
10 users sampled
20 users sampled
```

```
# Location of users in the test data

# Filtering raw_data to include only rows where the 'User_ID'
#is present in the te_users list (test set users)
test_plays = raw_data.loc[raw_data['User_ID'].isin(te_users)]

# Further filtering test_plays to include only rows where the 'Mov
#is present in the unique_sid list (unique movie IDs)
test_plays = test_plays.loc[test_plays['Movie_ID'].isin(unique_sic
```

```
# Splitting test data of movies for test set users into train and
# Using the previously defined function split_train_test_proportic
test_plays_tr, test_plays_te = split_train_test_proportion(test_pl
```

```
0 users sampled
10 users sampled
20 users sampled
```

Saving the data in user_index and item_index formats

```
# Function to numerize the data by mapping user and movie IDs to n
def numerize(data):
    # Mapping 'User_ID' values to corresponding numerical indices
    uid = list(map(lambda x: profile2id[x], data['User_ID']))
    # Mapping 'Movie_ID' values to corresponding numerical indices
    sid = list(map(lambda x: show2id[x],   data['Movie_ID']))
    # Creating a new DataFrame with numerical indices for users (
    return pd.DataFrame(data={'uid': uid, 'sid': sid}, columns=['u
```

```
# Converting the training data to numerical indices using the 'num
train_data = numerize(train_plays)

# Saving the training data to a CSV file named 'train.csv'
#in the 'pro_dir' directory without including the index
train_data.to_csv(os.path.join(pro_dir, 'train.csv'), index=False)
```

```
# Converting the validation training data to numerical indices usi
vad_data_tr = numerize(vad_plays_tr)

# Saving the validation training data to a CSV file named 'validat
#in the 'pro_dir' directory without including the index
vad_data_tr.to_csv(os.path.join(pro_dir, 'validation_tr.csv'), inc
```

```
# Converting the validation test data to numerical indices using t
vad_data_te = numerize(vad_plays_te)

# Saving the validation test data to a CSV file named 'validation_
#in the 'pro_dir' directory without including the index
vad_data_te.to_csv(os.path.join(pro_dir, 'validation_te.csv'), inc
```

```
# Converting the test training data to numerical indices using the
test_data_tr = numerize(test_plays_tr)

# Saving the test training data to a CSV file named 'test_tr.csv'
#in the 'pro_dir' directory without including the index
test_data_tr.to_csv(os.path.join(pro_dir, 'test_tr.csv'), index=Fa
```

```
# Converting the test test data to numerical indices using the 'nu
```

```
test_data_te = numerize(test_plays_te)

# Saving the test test data to a CSV file named 'test_te.csv'
#in the 'pro_dir' directory without including the index
test_data_te.to_csv(os.path.join(pro_dir, 'test_te.csv'), index=Fa
```

MODEL DEFINITION AND TRAINING

Notation: Let $u\in \{1,...,U\}$ be the indices of users and $i\in\{1,...,I\}$ be the indices of items. The user-item interaction matrix is $X\in\mathbb{N}^{U\times I}$. The interaction matrix is binarized.

Generative Process: For each user u, the model starts by sampling the K-dimensional latent representation $\mathbf{z}_u$ from a standard Gaussian prior. The latent representation $\mathbf{z}_u$ is transformed by a non-linear function $f_\theta (\cdot) \in \mathbb{R}^I$to produce a probability distribution over I items $\pi (\mathbf{z}_u)$ from which the interaction history $\mathbf{x}_u$ is assumed to be $$ \mathbf{z}_{u} \sim \mathscr{N}\left(0, \mathbf{I}_{K}\right), \pi\left(\mathbf{z}_{u}\right) \propto \exp \left\{f_{\theta}\left(\mathbf{z}_{u}\right)\}, \mathbf{x}_{u} \sim \operatorname{Mult}\left(N_{u}, \pi\left(\mathbf{z}_{u}\right)\right)\right. $$ The objective of Multi-DAE for a single user u is: $$ \mathscr{L}_{u}(\theta, \phi)=\log p_{\theta}\left(\mathbf{x}_{u} \mid g_{\phi}\left(\mathbf{x}_{u}\right)\right) $$ where $g_{\phi}(\cdot)$ is the non-linear "encoder" function.

- The *Saver* class adds operations to save and restore variables to and from checkpoints. It also offers convenient methods to execute these operations.
- tf.contrib.layers.xavier_initializer: Generates samples from a truncated normal distribution centered on 0 with standard deviation = sqrt(2 / (fan_in + fan_out)), where fan_in represents the number of input units in the weight tensor and fan_out represents the number of output units in the weight tensor.

Multinomial Blurring Autoencoder Model

Autoencoders: An autoencoder takes inputs $x\in[0,1]^d$ and first maps it (with an encoder) to a latent representation $h\in[0,1]^{d^\prime}$ through a deterministic mapping $$h=Wx+b,$$ where W represents the weight matrix and b the bias vector.

The latent representation h is then mapped back (with the decoder) into a reconstruction z of the same size as x through a similar transformation, i.e., $$z= (W^\prime h + b^\prime).$$ z is expected to be a prediction of x. The weight matrix $W^\prime$ for the inverse mapping can be constructed as $W^\prime=W^T$.

We aim to minimize $$\min_\theta\|x-z\|^2$$

building MultiDAE

```
##Clase de funciones para MULTINOMIAL DENOISING AUTOENCODERS (MULT
class MultiDAE(object):
    '''Arguments:
        p_dims = decoder dimension,
        q_dims = encoder dimension,
        lam = regularization parameter,
        lr = learning rate'''
    def __init__(self, p_dims, q_dims=None, lam=0.01, lr=1e-3, ran
         # Initializes the Multi-DAE class with decoder and encode
            #regularization parameter, learning rate, and random s
        self.p_dims = p_dims
        if q_dims is None:
            self.q_dims = p_dims[::-1]
        else:
            assert q_dims[0]  == p_dims[-1], "Input and output dim
            assert q_dims[-1] == p_dims[0], "Latent dimension for 
            self.q_dims = q_dims
        self.dims   = self.q_dims + self.p_dims[1:]

        self.lam = lam
        self.lr  = lr
        self.random_seed = random_seed

        self.construct_placeholders()

    def construct_placeholders(self):
        # Constructs placeholder variables for input data and drop
        tf.compat.v1.disable_eager_execution()
        self.input_ph     = tf.compat.v1.placeholder( dtype=tf.flo
        self.keep_prob_ph = tf.compat.v1.placeholder_with_default(
```

```
    #Latent Representation
    def forward_pass(self):
        # Performs the forward pass through the network architectu
        h = tf.compat.v1.nn.l2_normalize( self.input_ph, 1) #L2 no
        # Computes dropout: randomly sets elements to zero to prev
        h = tf.compat.v1.nn.dropout( h, self.keep_prob_ph ) #With
        #The remaining elements are scaled up by 1.0 / (1 - rate)
        #(To avoid overfitting) # construct vector y
        for i, (w,b) in enumerate(zip(self.weights, self.biases)):
            h = tf.compat.v1.matmul(h,w)+b #Latent Representation
            if i != len(self.weights)-1: #For the last layer.
                h = tf.compat.v1.nn.tanh(h) #Activation function
        return tf.compat.v1.train.Saver(), h

    def construct_weights(self):
        # Constructs weights and biases for the network layers
        self.weights = []
        self.biases  = []
        #I define weights.
        for i, (d_in,d_out) in enumerate(zip(self.dims[:-1],self.d
            weight_key = "Weight_{}to_{}".format(i,i+1) #Names
            bias_key   = "Bias_{}".format(i+1)
           #Creating new variables (weights) with TensorFlow
           #Weight matrix
            self.weights.append(tf.compat.v1.get_variable( name = 
                                                initializer = tf.keras.i
            #Bias vector
            self.biases.append(tf.compat.v1.get_variable( name = b
                                                initializer = tf.compat.

            #Adding statistical summary
            tf.compat.v1.summary.histogram( weight_key, self.weigh
            tf.compat.v1.summary.histogram( bias_key, self.biases[
    def loss(self):
        # Defines the loss function using negative log-likelihood
        with tf.GradientTape() as tape:
            with tf.compat.v1.Session() as sess:
                saver, logits = self.forward_pass()
                log_softmax_var = tf.compat.v1.nn.log_softmax(logi
                neg_ll = -tf.compat.v1.reduce_mean(tf.reduce_sum(l
                reg_var = tf.compat.v1.nn.l2_loss(self.lam)
                return neg_ll + 2.0 * reg_var
        train_op = tf.optimizers.Adam(self.lr).minimize(loss, var_
        return train_op
    def build_graph(self):
# Constructs the computational graph for
#the Multi-DAE model, including forward pass, loss calculation, op
        self.construct_weights()
```

```
        saver, logits = self.forward_pass()
        log_softmax_var = tf.compat.v1.nn.log_softmax(logits)
        neg_ll = -tf.compat.v1.reduce_mean(tf.compat.v1.reduce_sum
        reg_var = tf.add_n([ tf.nn.l2_loss(v) for v in self.weight
        print(np.shape(self.weights))
        loss    = neg_ll + 2.0 * reg_var
        train_op = tf.compat.v1.train.AdamOptimizer(self.lr).minim
        #train_op = tf.optimizers.Adam(self.lr).minimize(loss, var
        # add summary statistics
        tf.compat.v1.summary.scalar('negative_multi_ll', neg_ll)
        tf.compat.v1.summary.scalar('loss', loss)
        merged = tf.compat.v1.summary.merge_all()
        return saver, logits, loss, train_op, merged
```

Autoencoder Variacional Multinomial

The objective function of Multi-VAE ^{PR} (Evidence Lower Bound - ELBO) for a user u u is given by : $$ \mathcal{L}_u(\theta, \phi) = \mathbb{E}_{q_\phi(z_u | x_u)}[\log p_\theta(x_u | z_u)] - \beta \cdot KL(q_\phi(z_u | x_u) \| p(z_u)) $$ Here, q_ϕ is the approximate variational distribution (inference model), β is an additional control parameter. The dataset's objective function is the average over all users. Training can be performed similarly to Multi-DAE thanks to the reparameterization trick. The Kullback-Leibler divergence is defined as: $$ \operatorname{KL}\left(q\left(\mathrm{z}_{u}\right) \| p\left(\mathrm{z}_{u} \mid \mathrm{x}_{u}\right)\right)=\int_{-\infty}^{\infty} q\left(\mathrm{z}_{u}\right) \ln \frac{q\left(\mathrm{z}_{u}\right)}{p\left(\mathrm{z}_{u} \mid \mathrm{x}_{u}\right)} \mathrm{d} z_{u} $$

Assuming two distributions with parameters $q(z_u) \sim N\left(\mu_u,diag\{ \sigma^{2}_u\}\right)$ y $p(z_u) \sim N(0,I_K)$.

To provide further context in terms of latent variable models, the goal is to approximate a posterior to the true posterior by minimizing the KL divergence. Here, $ z_u $ represents the latent variable, $q(z_u)$ is the approximate distribution, and $ p(z_u) $ is the prior distribution. The parameters of q are the output from the encoder.

For Gaussian distributions, the KL divergence is given by:

$$ \begin{aligned} K L(q \| p) &=\frac{1}{2} \log (2 \pi)+\frac{1}{2}\left(\mu^{2}+\sigma^{2}\right)-\frac{1}{2} \log (2 \pi)-\frac{1}{2} \log \left(\sigma^{2}\right)-\frac{1}{2} \\ &=-\frac{1}{2}\left(1+\log \left(\sigma^{2}\right)+-\mu^{2}-\sigma^{2}\right) \end{aligned} $$ This equation

can be generalized for multivariate cases by summing across all dimensions, i.e., $$ K L(q \| p)=-\frac{1}{2} \sum_{d=1}^{D}\left(1+\log \left(\sigma_u^{2}\right)+-\mu_u^{2}-\sigma_u^{2}\right) $$ For more information on KL divergence, please refer to

https://leenashekhar.github.io/2019-01-30-KL-Divergence/

q_dims = dimension of the encoder layers $\mu=x_{ui}[:,:qdims[-1]]$, $qdims[-1]$ is the dimension of the last layer of the encoder.

$\log(var)=x_{ui}[:,qdims[-1]:]$

$\sigma = e^{0.5*\log(var)}$

Bulding MultiVEA that takes MultiDAE as input

```
class MultiVAE(MultiDAE):
    def construct_placeholders(self):#Building variables
        # Inherits and constructs placeholders from the parent cla
        super(MultiVAE, self).construct_placeholders() #This funct
        # Define additional placeholders with default values for s
         #preserve a method or attribute of a parent (parent) clas
         #without having to name it explicitly.
         # placeholders with default values when scoring
        self.is_training_ph = tf.compat.v1.placeholder_with_defaul
        self.anneal_ph = tf.compat.v1.placeholder_with_default(1.,

    def build_graph(self):
         # Constructs the computational graph for the MultiVAE moc
        self._construct_weights() # Constructs weights for encoder
        saver, logits, KL = self.forward_pass() # Performs forward
        log_softmax_var = tf.compat.v1.nn.log_softmax(logits) # Ac
        neg_ll = -tf.compat.v1.reduce_mean(tf.reduce_sum(log_softm
        #Aplicando regularización a los pesos
        reg_var = tf.add_n([ tf.nn.l2_loss(v) for v in (self.weigh
        # tensorflow l2 regularization multiply 0.5 to the l2 norm
        # multiply 2 so that it is back in the same scale
        neg_ELBO = neg_ll + self.anneal_ph * KL + 2.0 * reg_var  #
```

```
        # Optimizes parameters
        train_op = tf.compat.v1.train.AdamOptimizer(self.lr).minim

        # Summary statistics
        tf.compat.v1.summary.scalar('negative_multi_ll', neg_ll)
        tf.compat.v1.summary.scalar('KL', KL)
        tf.compat.v1.summary.scalar('neg_ELBO_train', neg_ELBO)
        merged = tf.compat.v1.summary.merge_all() # Merges all the

        return saver, logits, neg_ELBO, train_op, merged


# Encoder
    def q_graph(self):
        mu_q, std_q, KL = None, None, None
        h = tf.compat.v1.nn.l2_normalize( self.input_ph, 1)
        h = tf.compat.v1.nn.dropout( h, self.keep_prob_ph )
        print(h)
        for i, (w,b) in enumerate(zip(self.weights_q,self.biases_q
            #print("W", w)
            #print("b: ", b)
            #print("Hereee: ", tf.compat.v1.matmul(h,w)) #HERE IS
            h = tf.compat.v1.matmul(h,w) + b

            if i != len(self.weights_q)-1:
                h = tf.compat.v1.nn.tanh(h)

            else:
                #The dimension of the last encoder layer (q_dims)
                mu_q     = h[:,:self.q_dims[-1]] #Mean matrix xiu
                logvar_q = h[:,self.q_dims[-1]:]
                std_q    = tf.exp(0.5*logvar_q)
                KL       = tf.compat.v1.reduce_mean(tf.reduce_sum(
                                                              axis =
        return mu_q, std_q, KL

 # Decoder
    def p_graph(self,z):
        h = z
        for i, (w,b) in enumerate(zip(self.weights_p,self.biases_p
            h = tf.compat.v1.matmul(h,w)+b #Latent representation
            if i != len(self.weights_p)-1:
                h = tf.compat.v1.nn.tanh(h)
        return h
    def forward_pass(self): # Latent space calculation
        #q-network
        #print("here")
        mu_q, std_q, KL = self.q_graph()
```

```
        #print(tf.shape(std_q))
        epsilon         = tf.random.normal(tf.shape(std_q)) #Rando
        sampled_z       = mu_q + self.is_training_ph*\
            epsilon*std_q #Reparametrization
        #p-network
        logits = self.p_graph(sampled_z)
        return tf.compat.v1.train.Saver(), logits, KL #train.Saver


    def _construct_weights(self):
         # Constructs weights and biases for the encoder
        self.weights_q = []
        self.biases_q  = []
        for i, (d_in, d_out) in enumerate(zip(self.q_dims[:-1],sel
            if i == len(self.q_dims[:-1])-1:#penúltima capa para e
                d_out *= 2
            weight_key = "Weight_q_{}_to{}".format(i,i+1)
            bias_key   = "Bias_q_{}".format(i+1)

            self.weights_q.append(tf.compat.v1.get_variable( name
                                initializer = tf.keras.initializers.gl
            self.biases_q.append(tf.compat.v1.get_variable( name =
                                initializer = tf.compat.v1.truncated_r

        # Adding statistical summary
        tf.compat.v1.summary.histogram(weight_key, self.weights_q[
        tf.compat.v1.summary.histogram(bias_key, self.biases_q[-1]

         # Constructs weights and biases for the decoder
        self.weights_p = []
        self.biases_p  = []
        for i, (d_in, d_out) in enumerate(zip(self.p_dims[:-1],sel
            weight_key = "Weight_p_{}_to_{}".format(i,i+1)
            bias_key   = "Bias_p_{}".format(i+1)

            self.weights_p.append(tf.compat.v1.get_variable( name
                                initializer = tf.keras.initializers.gl
            self.biases_p.append(tf.compat.v1.get_variable( name =
                                initializer = tf.compat.v1.truncated_r

        # Adding statistical summary
        tf.compat.v1.summary.histogram(weight_key, self.weights_p[
        tf.compat.v1.summary.histogram(bias_key, self.biases_p[-1]
```

Training and Validation of Data

We load the preprocessed training and validation data

```
unique_sid = list()# Initializes an empty list to store unique ide
#Movie data (indexes)

# Reading movie data (indices) from a file and populating the uniq
with open(os.path.join(pro_dir, 'unique_sid.txt'), 'r') as f:
    for line in f:
        # Appends each line (movie index) after stripping any lead
        unique_sid.append(line.strip())
    # Calculates the number of items by determining the length of
    #(number of unique movie indices)
n_items = len(unique_sid)
```

```
# Function to load training data from a file
def load_train_data(file):
    dat     = pd.read_csv(file)# Reads data from the specified fil
    n_users = dat['uid'].max()+1# Calculates the number of users b

    rows    = dat['uid'] # Selects the 'uid' column of the data
    cols    = dat['sid'] # Selects the 'sid' (movie index) column

    # Constructs a sparse matrix with user-item interactions
    data    = sparse.csr_matrix((np.ones_like(rows),(rows, cols)),
                    dtype = 'float64', shape=(n_users,n_items))

    return data # Returns the constructed sparse matrix represent
```

```
# Loading training data by calling the previously defined functio
# The function reads the 'train.csv' file located in the 'pro_dir'
train_data = load_train_data(os.path.join(pro_dir, 'train.csv'))
```

```
type(train_data) # Determines and displays the data type of the va
```

```
scipy.sparse.csr.csr_matrix
```

```
# Function to load validation data
def load_tr_te_data(file_tr,file_te):# Reading data from training
    dat_tr = pd.read_csv(file_tr) # Loading data from training fil
    dat_te = pd.read_csv(file_te) # Loading data from testing file
    # Range of item indices
    start_idx = min(dat_tr['uid'].min(),dat_te['uid'].min())# Finc
    end_idx   = max(dat_tr['uid'].max(),dat_te['uid'].max())# Finc

    # Creating row and column indices for training data
    rows_train = dat_tr['uid']-start_idx
    cols_train = dat_tr['sid']

    # Creating row and column indices for testing data
    rows_test  = dat_te['uid']-start_idx
    cols_test  = dat_te['sid']

    # Creating a sparse matrix for training data
    data_train = sparse.csr_matrix((np.ones_like(rows_train),(rows
                                    dtype = 'float64', shape=(end_idx-s
     # Creating a sparse matrix for testing data
    data_test  = sparse.csr_matrix((np.ones_like(rows_test),(rows_
                                    dtype = 'float64', shape=(end_idx-s
    return data_train, data_test # Returning the generated trainir
```

```
# Loading validation data by calling the function load_tr_te_data
# Loading training and testing data files from specified paths usi
val_data_train,val_data_test = load_tr_te_data(os.path.join(pro_di
                                                os.path.join(pro_di
```

```
type(val_data_test)# Checking the data type of the variable val_dc
```

```
scipy.sparse.csr.csr_matrix
```

Training hyperparameters:

```
# Slicing the train_data to the first 10,000 rows and all columns
train_data = train_data[:10000,:]

# Obtaining the number of rows (observations) in the train_data
```

```
N = train_data.shape[0]  # 115308 observations initially, but only

# Creating a range of indices from 0 to N-1
idxlist = range(N) # Range of indices for the train_data

# Determining batch size for training
batch_size = 500

# Calculating the number of batches per epoch based on the batch s
val = np.ceil(float(N)/batch_size) # Calculating the number of bat
batches_per_epoch = int(val) # Converting to an integer using np.c

# Obtaining the number of observations in the validation data (val
N_vad = val_data_train.shape[0]  # Number of observations in the v

# Creating a range of indices for the validation dataset
idxlist_vad = range(N_vad) # Range of indices for the validation c

# Setting batch size for validation
batch_size_vad = 20000 # Batch size for validation data

# Total number of gradient updates for annealing
# prevent the model from getting stuck in local minima and to impr
total_anneal_steps = 2000000 # Total steps for gradient updates fc

# Annealing parameter (maximum value)
# As the annealing process progresses, this parameter is likely tc
anneal_cap = 0.9 # Annealing parameter, a maximum value used in th
```

Evaluate function:

Normalized discounted cumulative gain (NDCG@k) and Recall@k

Discounted Cumulative Gain (DCG) is the metric to measure the quality of the classification. It is mainly used in information retrieval problems, such as measuring the effectiveness of the search engine's algorithm by ranking the articles it displays according to their relevance in terms of the search keyword.

```
def NDCG_binary_at_k_batch(X_pred, heldout_batch, k):
```

```
    '''
    normalized discounted cumulative gain@k for binary relevance
    ASSUMPTIONS: all the 0's in heldout_data indicate 0 relevance
    '''
    # Number of users in the batch
    batch_users = X_pred.shape[0]

    # Finding the indices partitioned to the top-k values
    idx_topk_part = bn.argpartition(-X_pred, k, axis=1)

    # Extracting the top-k values for each user
    topk_part = X_pred[np.arange(batch_users)[:, np.newaxis],
                       idx_topk_part[:, :k]]

    # Sorting the indices of top-k values
    idx_part = np.argsort(-topk_part, axis=1)

    # Sorting the top-k predicted scores
    idx_topk = idx_topk_part[np.arange(batch_users)[:, np.newaxis]

    # Building the discount template based on the position
    tp = 1. / np.log2(np.arange(2, k + 2))

    # Calculating Discounted Cumulative Gain (DCG)
    DCG = (heldout_batch[np.arange(batch_users)[:, np.newaxis],
                         idx_topk].toarray() * tp).sum(axis=1)

    # Calculating Ideal Discounted Cumulative Gain (IDCG)
    IDCG = np.array([(tp[:min(n, k)]).sum()
                     for n in heldout_batch.getnnz(axis=1)])
    return DCG / IDCG
```

```
def Recall_at_k_batch(X_pred, heldout_batch, k=100):
    batch_users = X_pred.shape[0]

    idx = bn.argpartition(-X_pred, k, axis=1)
    X_pred_binary = np.zeros_like(X_pred, dtype=bool)
    X_pred_binary[np.arange(batch_users)[:, np.newaxis], idx[:, :k

    X_true_binary = (heldout_batch > 0).toarray()
    tmp = (np.logical_and(X_true_binary, X_pred_binary).sum(axis=1
        np.float32)
    recall = tmp / np.minimum(k, X_true_binary.sum(axis=1))
    return recall
```

Multi-DAE Training & Validation

multilayer perceptron (MLP) with symmetrical architecture. The generative function is a [200 -> 600 -> n_items] MLP, which means the inference function is a [n_items -> 600 -> 200] MLP. Thus the overall architecture for the Multi-VAE^{PR} is [n_items -> 600 -> 200 -> 600 -> n_items].

```
# Defining the dimensions for the neural network layers
# The first layer has 200 units/neurons and represents the latent
# The second layer represents the number of items in the dataset
p_dims = [200, n_items]
#Defining Output dimensions for the neural network layers
q_dims = [n_items, 600, 200]
```

```
# Resetting the default TensorFlow graph
tf.compat.v1.reset_default_graph()

# Creating an instance of the MultiDAE class with specified parame
# p_dims: dimensions for the neural network layers
# lam: regularization parameter for the model
# random_seed: seed for random number generation
dae = MultiDAE(p_dims, lam=0.01 / batch_size, random_seed=98765)

# Building the TensorFlow computation graph for the DAE model
saver, logits_var, loss_var, train_op_var, merged_var = dae.build_

# Creating a TensorFlow variable initialized with a value of 0.0 f
ndcg_var = tf.Variable(0.0)

# Placeholder for NDCG values of type float64 with unspecified sha
ndcg_dist_var = tf.compat.v1.placeholder(dtype=tf.float64, shape=N

# Creating a summary for NDCG at k for validation
ndcg_summary = tf.summary.scalar('ndcg_at_k_validation', ndcg_var)

# Creating a summary for the histogram of NDCG at k for validation
ndcg_dist_summary = tf.summary.histogram('ndcg_at_k_hist_validatio

# Merging the NDCG summary and histogram summaries into a single s
# merged_valid = tf.compat.v1.summary.merge([ndcg_summary, ndcg_di
```

```
(2,)
```

```
ndcg_dist_summary# Outputting the summary for the histogram of NDC
print(ndcg_dist_summary)
print(logits_var)# Printing the variable 'logits_var'
```

```
Tensor("ndcg_at_k_hist_validation/write_summary/Const:0", shape=
Tensor("add_1:0", shape=(None, 219), dtype=float32)
```

```
n_epochs = 200 # Setting the number of epochs for training to 200
```

```
arch_str = "I-%s-I" % ('-'.join([str(d) for d in dae.dims[1:-1]]))
```

```
chkpt_dir = '/volmount/chkpt/ml-20m/DAE/{}'.format(arch_str)
if not os.path.isdir(chkpt_dir):
    os.makedirs(chkpt_dir)

print("chkpt directory: %s" % chkpt_dir)
```

```
chkpt directory: /volmount/chkpt/ml-20m/DAE/I-200-I
```

```
ndcgs_vad = []# Empty list to store NDCG values for validation
i=0 # Counter variable initialization
n100_list, r20_list, r50_list = [], [], []
# TensorFlow session context manager
with tf.compat.v1.Session() as sess:
# Initializing global variables within the session
    init = tf.compat.v1.global_variables_initializer()
    sess.run(init)

# Initializing the best NDCG variable to negative infinity
    best_ndcg = -np.inf

    # Looping through the specified number of epochs for training
    for epoch in range(n_epochs):
        i+=1 # Incrementing the counter

        # Shuffling the index list for training
        idxlist = random.sample(list(idxlist), len(list(idxlist)))
```

```
        # Training for one epoch
        for bnum, st_idx in enumerate(range(0, N, batch_size)):
            end_idx = min(st_idx + batch_size, N)
            X = train_data[idxlist[st_idx:end_idx]]
            # Preparing input data for training

            if sparse.isspmatrix(X):
                X = X.toarray()
            X = X.astype('float32')

            # Creating a feed dictionary for the model training
            feed_dict = {dae.input_ph: X,
                         dae.keep_prob_ph: 0.5}

            # Running the training operation for the model
            sess.run(train_op_var, feed_dict=feed_dict)

            if bnum % 100 == 0:
                summary_train = sess.run(merged_var, feed_dict=fee
                #summary_writer.add_summary(summary_train, global_

        # Computing validation NDCG
        ndcg_dist = []
        for bnum, st_idx in enumerate(range(0, N_vad, batch_size_v
            end_idx = min(st_idx + batch_size_vad, N_vad)
#            print()
            X = val_data_train[idxlist_vad[st_idx:end_idx]]
            #print("X",X)
            if sparse.isspmatrix(X):
                X = X.toarray()
            X = X.astype('float32')

            # Getting predictions for validation data
            pred_val = sess.run(logits_var, feed_dict={dae.input_p

            # Excluding examples from training and validation (if
            pred_val[X.nonzero()] = -np.inf
            #print("Validation: ",NDCG_binary_at_k_batch(pred_val,
            #print(pred_val)
            #Calculating NDCG for validation data
            ndcg_dist.append(NDCG_binary_at_k_batch(pred_val, val_
            r20_list.append(Recall_at_k_batch(pred_val, val_data_t
            r50_list.append(Recall_at_k_batch(pred_val, val_data_t
            #print("List: ", ndcg_dist)

        print("Iteration: ", i )# Printing iteration and mean NDC(
        ndcg_dist = np.concatenate(ndcg_dist)
```

```
        ndcg_dist = np.nan_to_num(ndcg_dist)
        ndcg_ = ndcg_dist.mean() * 100
        print("Mean NDCG: ", ndcg_)

         # Appending the mean NDCG to the list
        ndcgs_vad.append(ndcg_)


r20_list = np.concatenate(r20_list)
r50_list = np.concatenate(r50_list)
```

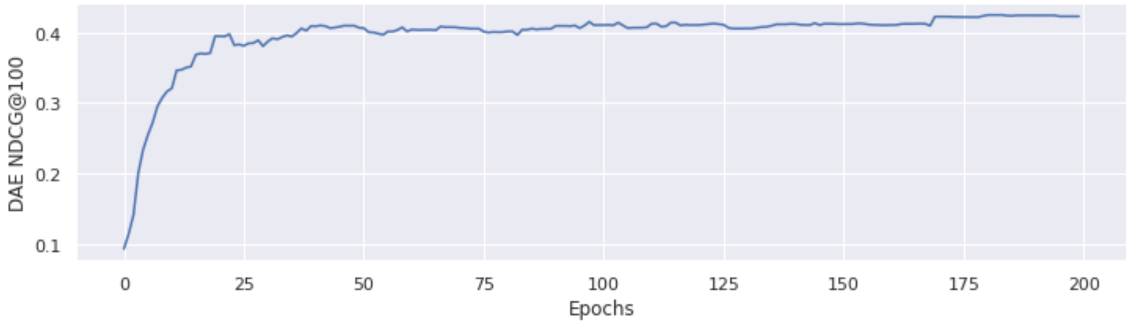
```
Iteration:  1
Mean NDCG:  0.09330675132699187
Iteration:  2
Mean NDCG:  0.11444963009987835


Iteration:  3
Mean NDCG:  0.1411779941215017
Iteration:  4
Mean NDCG:  0.20059498314589516
Iteration:  5
Mean NDCG:  0.23355260371425768
Iteration:  6
Mean NDCG:  0.2544972791528825
Iteration:  7
Mean NDCG:  0.27215390319446914
Iteration:  8
Mean NDCG:  0.29519160555860396
Iteration:  9
Mean NDCG:  0.30720563323076494


/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:32:
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:11:
  # This is added back by InteractiveShellApp.init_path()
```

```
# Creating a new figure for plotting with specified size (width: 1
plt.figure(figsize=(12, 3))

# Plotting the values of NDCG for validation data across epochs
plt.plot(ndcgs_vad)

# Labeling the y-axis as "Validation NDCG@100"
plt.ylabel("DAE NDCG@100")
```

```
# Labeling the x-axis as "Epochs"
plt.xlabel("Epochs")
pass
```

<Figure size 864x216 with 1 Axes>

Download




```
print("DAE Recall@20 = " , (np.nanmean(r20_list)))
print("DAE Recall@50 = " ,(np.nanmean(r50_list)))
```

```
DAE Recall@20 =  0.376
DAE Recall@50 =  0.63803125
```

MultiVAE Traing & Validation

```
# Resetting the default TensorFlow graph
tf.compat.v1.reset_default_graph()

# Defining the dimensions for the neural network layers [200, 600,
p_dims = [200, 600, n_items]

# Creating an instance of the MultiVAE class with specified parame
vae = MultiVAE(p_dims, lam=0.01, random_seed=98765)

# Building the TensorFlow computation graph for the VAE model
saver, logits_var, loss_var, train_op_var, merged_var = vae.build_

# Creating a TensorFlow variable initialized with a value of 0.0 f
ndcg_var = tf.Variable(0.0)

# Placeholder for NDCG values of type float64 with unspecified sha
```

```python
ndcg_dist_var = tf.compat.v1.placeholder(dtype=tf.float64, shape=N

# Creating a summary for NDCG at k for validation
ndcg_summary = tf.summary.scalar('ndcg_at_k_validation', ndcg_var)

# Creating a summary for the histogram of NDCG at k for validation
ndcg_dist_summary = tf.summary.histogram('ndcg_at_k_hist_validatio

#merged_valid = tf.compat.v1.summary.merge([ndcg_summary, ndcg_dis
```

```
Tensor("dropout/Mul:0", shape=(None, 219), dtype=float32)
```

```python
n_epochs = 200 # Setting the number of epochs for training the moc
```

```python
ndcgs_vad = []# Empty list to store NDCG values for validation
i=0# Counter variable initialization
n100_list, r20_list, r50_list = [], [], []
with tf.compat.v1.Session() as sess:# TensorFlow session context m
    # Initializing global variables within the session
    init = tf.compat.v1.global_variables_initializer()
    sess.run(init)

    best_ndcg = -np.inf # Initializing the best NDCG variable to n

    update_count = 0.0 # Initializing update_count for annealing

    # Training for one epoch
    for epoch in range(n_epochs):# Looping through the specified n
        i+=1# Incrementing the counter
        idxlist = random.sample(list(idxlist), len(list(idxlist)))
        # train for one epoch
        for bnum, st_idx in enumerate(range(0, N, batch_size)):
            end_idx = min(st_idx + batch_size, N)
            X = train_data[idxlist[st_idx:end_idx]]# Preparing inp

            if sparse.isspmatrix(X):
                X = X.toarray()
            X = X.astype('float32')

            # Annealing schedule
            if total_anneal_steps > 0:
                anneal = min(anneal_cap, 1. * update_count / total
            else:
                anneal = anneal_cap
```

```
            # Creating a feed dictionary for the model training
            feed_dict = {vae.input_ph: X,
                         vae.keep_prob_ph: 0.5,
                         vae.anneal_ph: anneal,
                         vae.is_training_ph: 1}
            sess.run(train_op_var, feed_dict=feed_dict) # Running

            if bnum % 100 == 0:
                summary_train = sess.run(merged_var, feed_dict=fee
            update_count += 1

         # Computing validation NDCG
        ndcg_dist = []
        for bnum, st_idx in enumerate(range(0, N_vad, batch_size_v
            end_idx = min(st_idx + batch_size_vad, N_vad)
            X = val_data_train[idxlist_vad[st_idx:end_idx]]
            if sparse.isspmatrix(X):
                X = X.toarray()
            X = X.astype('float32')

          # Getting predictions for validation data
            pred_val = sess.run(logits_var, feed_dict={vae.input_p
                         vae.anneal_ph: anneal,
                         vae.is_training_ph: 1} )
            # exclude examples from training and validation (if ar
            pred_val[X.nonzero()] = -np.inf
            ndcg_dist.append(NDCG_binary_at_k_batch(pred_val, val_
            r20_list.append(Recall_at_k_batch(pred_val, val_data_t
            r50_list.append(Recall_at_k_batch(pred_val, val_data_t


        print("Iteración: ", i ) # Printing iteration and mean NDC
        ndcg_dist = np.concatenate(ndcg_dist)
        ndcg_dist = np.nan_to_num(ndcg_dist)
        ndcg_ = ndcg_dist.mean() * 100
        print("Mean: ", ndcg_)
        ndcgs_vad.append(ndcg_)# Appending the mean NDCG to the li
        # update the best model (if necessary)
        if ndcg_ > best_ndcg:
            saver.save(sess, '{}/model'.format(chkpt_dir))
            best_ndcg = ndcg_



r20_list = np.concatenate(r20_list)
r50_list = np.concatenate(r50_list)
```

```
Iteración:  1
Mean:  0.2083989171888481
Iteración:  2
Mean:  0.1091639100091735
Iteración:  3
Mean:  0.13722700295174967
Iteración:  4
Mean:  0.17227502242020296
Iteración:  5
Mean:  0.13558848571411336
Iteración:  6
Mean:  0.2407723443036775
Iteración:  7
Mean:  0.18706706251435007
Iteración:  8
Mean:  0.2693704645729751
Iteración:  9
Mean:  0.27785853958770523
```

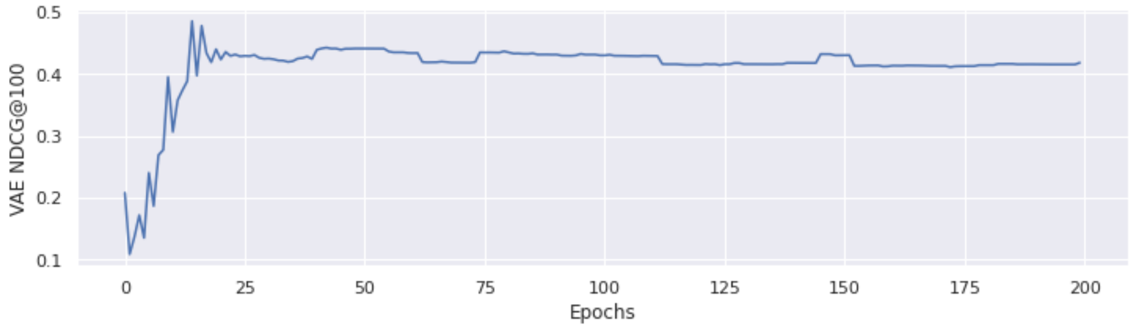
```
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:32:
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:11:
  # This is added back by InteractiveShellApp.init_path()
```

Measurement of average algorithm performance:

```
plt.figure(figsize=(12, 3))# Creating a new figure for plotting wi
plt.plot(ndcgs_vad)# Plotting the values of NDCG for validation da
plt.ylabel("VAE NDCG@100")# Labeling the y-axis as "Validation NDC
plt.xlabel("Epochs")# Labeling the x-axis as "Epochs"
pass
```

```
<Figure size 864x216 with 1 Axes>
```

Download




```
print("VAE Recall@20 = " , (np.nanmean(r20_list)))
print("VAE Recall@50 = " ,(np.nanmean(r50_list)))
```

```
VAE Recall@20 =  0.37096874999999996
VAE Recall@50 =  0.6079999999999999
```

Testing

Load the test data and compute test metrics

```
test_data_tr, test_data_te = load_tr_te_data(
    os.path.join(pro_dir, 'test_tr.csv'),
    os.path.join(pro_dir, 'test_te.csv'))
```

```
N_test = test_data_tr.shape[0]
idxlist_test = range(N_test)
batch_size_test = 2000
```

```
# Resetting the default TensorFlow graph
tf.compat.v1.reset_default_graph()
vae = MultiVAE(p_dims, lam=0.0)
saver, logits_var, _, _, _ = vae.build_graph()
```

```
Tensor("dropout/Mul:0", shape=(None, 219), dtype=float32)
```

Load the best performing model on the validation set

```
chkpt_dir = '/volmount/chkpt/ml-20m/DAE/{}'.format(arch_str)
print("chkpt directory: %s" % chkpt_dir)
```

```
chkpt directory: /volmount/chkpt/ml-20m/DAE/I-200-I
```

```
n100_list, r20_list, r50_list = [], [], []

# TensorFlow session context manager
with tf.compat.v1.Session() as sess:
    saver.restore(sess, '{}/model'.format(chkpt_dir))

    for bnum, st_idx in enumerate(range(0, N_test, batch_size_test
        end_idx = min(st_idx + batch_size_test, N_test)
        X = test_data_tr[idxlist_test[st_idx:end_idx]]

        if sparse.isspmatrix(X):
            X = X.toarray()
        X = X.astype('float32')

        pred_val = sess.run(logits_var, feed_dict={vae.input_ph: X
        # exclude examples from training and validation (if any)
        pred_val[X.nonzero()] = -np.inf
        n100_list.append(NDCG_binary_at_k_batch(pred_val, test_dat
        r20_list.append(Recall_at_k_batch(pred_val, test_data_te[i
        r50_list.append(Recall_at_k_batch(pred_val, test_data_te[i

n100_list = np.concatenate(n100_list)
r20_list = np.concatenate(r20_list)
r50_list = np.concatenate(r50_list)
```

```
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:32:
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:11:
  # This is added back by InteractiveShellApp.init_path()
```

```
print("Test NDCG@100 = " , (np.nanmean(n100_list)))
print("Test Recall@20 = " , (np.nanmean(r20_list)))
print("Test Recall@50 = " ,(np.nanmean(r50_list)))
```

```
Test NDCG@100 =  0.1777299186717457
```

```
Test Recall@20 =  0.125
Test Recall@50 =  0.5
```

Results

compared to the paper results we have slight enhancment sinice

- we have got 0.431 NDCG at Multi-DAE
- while we have got 0.52 at Multi-VAE.

Intuitively, Mult-vae and Mult-dae have roughly similar performance. but Mult-vae imposes stronger modeling assumptions and therefore could be more robust when user-item interaction data is scarce.

Future Work

To prevent Bias we used an undersampling technique that handles imbalanced data to prevent Variance we used Regularization L2 and to prevent overfitting but we still gained low NDCG so in the future, we want to Increase the size of the dataset to prevent underfiting or use simpler models that are less prone to overfitting, especially when the amount of data is limited.

Conclusion:

In the end, our journey through this recommendation system course project has been an enriching experience. We have not only explored a research paper and implemented its findings but have also gone above and beyond, striving to improve and innovate within the field. Through our collective efforts, dedication, and unwavering commitment, we hope to contribute to the advancement of recommendation systems and leave a lasting impact on the field as aspiring researchers and engineers.