1M

April 1, 2025

0.1 *Assignment 2*

[1]:
```
import torch
torch.cuda.empty_cache()
```

Import neccessary libraries and load the ratings

[2]:
```
import pandas as pd
import numpy as np

# Cargar el archivo ratings.dat especificando el separador '::'
ratings = pd.read_csv("../../data/ml-1m/ratings.dat", sep="::",␣
↪engine="python", header=None,
                      names=["UserID", "MovieID", "Rating", "Timestamp"])

# Mostrar las primeras filas para verificar la carga
print(ratings.head())
print(ratings.shape)
print(ratings.columns)

# Filtro: usuarios con al menos 5 ratings
user_counts = ratings['UserID'].value_counts()
ratings = ratings[ratings['UserID'].isin(user_counts[user_counts >= 5].index)]

# Filtro: películas con al menos 5 ratings
movie_counts = ratings['MovieID'].value_counts()
ratings = ratings[ratings['MovieID'].isin(movie_counts[movie_counts >= 5].
↪index)]

print(f"Usuarios después de filtrar: {ratings['UserID'].nunique()}")
print(f"Películas después de filtrar: {ratings['MovieID'].nunique()}")
```

```
   UserID  MovieID  Rating  Timestamp
0       1     1193       5  978300760
1       1      661       3  978302109
2       1      914       3  978301968
3       1     3408       4  978300275
4       1     2355       5  978824291
(1000209, 4)
```

```
Index(['UserID', 'MovieID', 'Rating', 'Timestamp'], dtype='object')
Usuarios después de filtrar: 6040
Películas después de filtrar: 3416
```

[3]:
```python
from sklearn.preprocessing import LabelEncoder

# Mapeo de IDs con LabelEncoder (más ordenado y reutilizable)
user_encoder = LabelEncoder()
movie_encoder = LabelEncoder()

ratings['userIndex'] = user_encoder.fit_transform(ratings['UserID'])
ratings['movieIndex'] = movie_encoder.fit_transform(ratings['MovieID'])

# Normalización a [0, 1]
ratings['rating_norm'] = ratings['Rating'] / 5.0

# Estandarización (media 0, desviación 1)
mean_rating = ratings['Rating'].mean()
std_rating = ratings['Rating'].std()
ratings['rating_std'] = (ratings['Rating'] - mean_rating) / std_rating

# Mostrar resumen
print(ratings[['Rating', 'rating_norm', 'rating_std']].head())
print(f"Media original: {mean_rating:.4f} | Desviación: {std_rating:.4f}")
```

```
   Rating  rating_norm  rating_std
0       5          1.0    1.269615
1       3          0.6   -0.521065
2       3          0.6   -0.521065
3       4          0.8    0.374275
4       5          1.0    1.269615
Media original: 3.5820 | Desviación: 1.1169
```

[ ]:
```python
# Convertir timestamp a datetime
ratings['datetime'] = pd.to_datetime(ratings['Timestamp'], unit='s')

# Extraer año, mes y día de la semana
ratings['year'] = ratings['datetime'].dt.year
ratings['month'] = ratings['datetime'].dt.month
ratings['dayofweek'] = ratings['datetime'].dt.dayofweek  # 0=Lunes, 6=Domingo

# Mostrar resumen
print(ratings[['Timestamp', 'datetime', 'year', 'month', 'dayofweek']].head())
```

```
   Timestamp            datetime  year  month  dayofweek
0  978300760 2000-12-31 22:12:40  2000     12          6
1  978302109 2000-12-31 22:35:09  2000     12          6
2  978301968 2000-12-31 22:32:48  2000     12          6
3  978300275 2000-12-31 22:04:35  2000     12          6
```

```
4  978824291 2001-01-06 23:38:11  2001      1          5
```

Import neccessary libraries and load the movies

```
[7]: import pandas as pd

# Cargar el archivo movies.dat con codificación ISO-8859-1 y separador '::'
movies = pd.read_csv("../../data/ml-1m/movies.dat", sep="::", engine="python",␣
 ↪header=None,
                     names=["MovieID", "Title", "Genres"], encoding="latin1")

# Mostrar las primeras filas para verificar la carga
print(movies.head())
print(movies.shape)
print(movies.columns)
```

```
   MovieID                               Title                        Genres
0        1                    Toy Story (1995)   Animation|Children's|Comedy
1        2                      Jumanji (1995)  Adventure|Children's|Fantasy
2        3             Grumpier Old Men (1995)                Comedy|Romance
3        4            Waiting to Exhale (1995)                  Comedy|Drama
4        5  Father of the Bride Part II (1995)                        Comedy
(3883, 3)
Index(['MovieID', 'Title', 'Genres'], dtype='object')
```

Unir ratings y Movies

```
[8]: # Unir ratings con movies usando MovieID
ratings = ratings.merge(movies, on="MovieID", how="inner")

# Mostrar algunas columnas nuevas para comprobar
print(ratings[['MovieID', 'Title', 'Genres']].head())
```

```
   MovieID                                   Title  \
0     1193  One Flew Over the Cuckoo's Nest (1975)
1      661        James and the Giant Peach (1996)
2      914                     My Fair Lady (1964)
3     3408                  Erin Brockovich (2000)
4     2355                    Bug's Life, A (1998)

                         Genres
0                         Drama
1  Animation|Children's|Musical
2               Musical|Romance
3                         Drama
4   Animation|Children's|Comedy
```

one hot encoding para los generos

[9]:
```python
# Separar los géneros por '|' y obtener one-hot encoding
genres_onehot = ratings['Genres'].str.get_dummies(sep='|')

# Añadir los géneros al dataframe de ratings
ratings = pd.concat([ratings, genres_onehot], axis=1)

# Mostrar las columnas de género
print(genres_onehot.columns.tolist())
print(ratings[['Title'] + genres_onehot.columns.tolist()].head(3))
```

```
['Action', 'Adventure', 'Animation', "Children's", 'Comedy', 'Crime',
'Documentary', 'Drama', 'Fantasy', 'Film-Noir', 'Horror', 'Musical', 'Mystery',
'Romance', 'Sci-Fi', 'Thriller', 'War', 'Western']
                                    Title  Action  Adventure  Animation  \
0  One Flew Over the Cuckoo's Nest (1975)       0          0          0
1        James and the Giant Peach (1996)       0          0          1
2                     My Fair Lady (1964)       0          0          0

   Children's  Comedy  Crime  Documentary  Drama  Fantasy  Film-Noir  Horror  \
0           0       0      0            0      1        0          0       0
1           1       0      0            0      0        0          0       0
2           0       0      0            0      0        0          0       0

   Musical  Mystery  Romance  Sci-Fi  Thriller  War  Western
0        0        0        0       0         0    0        0
1        1        0        0       0         0    0        0
2        1        0        1       0         0    0        0
```

Extraer los generos

[10]:
```python
import torch

genre_columns = genres_onehot.columns.tolist()

# Extraer vectores de género para cada entrada
genre_vectors = torch.tensor(ratings[genre_columns].values, dtype=torch.float32)

# Mostrar ejemplo
print(genre_vectors.shape)
print(genre_vectors[:3])
```

```
torch.Size([999611, 18])
tensor([[0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.],
        [0., 0., 1., 1., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.,
0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 1., 0., 0., 0.,
0.]])
```

dataset personalizado

[11]:
```
from torch.utils.data import Dataset

class MovieLensDataset(Dataset):
    def __init__(self, users, movies, genres, ratings):
        self.users = users
        self.movies = movies
        self.genres = genres
        self.ratings = ratings

    def __len__(self):
        return len(self.ratings)

    def __getitem__(self, idx):
        return {
            'user': self.users[idx],
            'movie': self.movies[idx],
            'genre': self.genres[idx],
            'rating': self.ratings[idx]
        }
```

[13]:
```
from sklearn.model_selection import train_test_split

# Elegimos columna de rating
rating_col = 'rating_norm'  # o 'rating_std'

# Filtrar usuarios con al menos 3 ratings
user_counts = ratings['UserID'].value_counts()
ratings_filtered = ratings[ratings['UserID'].isin(user_counts[user_counts >= 3].
 ↪index)]

# Nuevo split
train_list, val_list, test_list = [], [], []

for user_id, group in ratings_filtered.groupby('UserID'):
    user_train, user_temp = train_test_split(group, test_size=0.30,␣
 ↪random_state=42)
    user_val, user_test = train_test_split(user_temp, test_size=0.50,␣
 ↪random_state=42)
    train_list.append(user_train)
    val_list.append(user_val)
    test_list.append(user_test)

train_data = pd.concat(train_list).reset_index(drop=True)
val_data = pd.concat(val_list).reset_index(drop=True)
test_data = pd.concat(test_list).reset_index(drop=True)
```

```
# Extraer tensores (incluyendo géneros ahora sí)
train_user = torch.tensor(train_data['userIndex'].values, dtype=torch.long)
train_movie = torch.tensor(train_data['movieIndex'].values, dtype=torch.long)
train_rating = torch.tensor(train_data[rating_col].values, dtype=torch.float32)
train_genres = torch.tensor(train_data[genre_columns].values, dtype=torch.
 ↪float32)

val_user = torch.tensor(val_data['userIndex'].values, dtype=torch.long)
val_movie = torch.tensor(val_data['movieIndex'].values, dtype=torch.long)
val_rating = torch.tensor(val_data[rating_col].values, dtype=torch.float32)
val_genres = torch.tensor(val_data[genre_columns].values, dtype=torch.float32)

test_user = torch.tensor(test_data['userIndex'].values, dtype=torch.long)
test_movie = torch.tensor(test_data['movieIndex'].values, dtype=torch.long)
test_rating = torch.tensor(test_data[rating_col].values, dtype=torch.float32)
test_genres = torch.tensor(test_data[genre_columns].values, dtype=torch.float32)
```

[17]:
```
# Dataset personalizado ya lo definiste antes (con genre incluido)

train_dataset = MovieLensDataset(train_user, train_movie, train_genres,␣
 ↪train_rating)
val_dataset = MovieLensDataset(val_user, val_movie, val_genres, val_rating)
test_dataset = MovieLensDataset(test_user, test_movie, test_genres, test_rating)
```

[18]:
```
from torch.utils.data import DataLoader

batch_size = 512  # Puedes ajustarlo según tu GPU

train_loader = DataLoader(
    train_dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=4,
    pin_memory=True
)

val_loader = DataLoader(
    val_dataset,
    batch_size=batch_size,
    shuffle=False,
    num_workers=4,
    pin_memory=True
)

test_loader = DataLoader(
    test_dataset,
    batch_size=batch_size,
```

```
    shuffle=False,
    num_workers=4,
    pin_memory=True
)
```

modelo

[19]:
```
import torch.nn as nn
import torch.nn.functional as F

class NeuMF(nn.Module):
    def __init__(self, num_users, num_movies, num_genres, embedding_dim_gmf=32,␣
 ↪embedding_dim_mlp=32, dropout=0.3):
        super().__init__()

        # Embeddings GMF
        self.user_embedding_gmf = nn.Embedding(num_users, embedding_dim_gmf)
        self.movie_embedding_gmf = nn.Embedding(num_movies, embedding_dim_gmf)

        # Embeddings MLP
        self.user_embedding_mlp = nn.Embedding(num_users, embedding_dim_mlp)
        self.movie_embedding_mlp = nn.Embedding(num_movies, embedding_dim_mlp)

        # Mini-MLP para géneros
        self.genre_layer = nn.Sequential(
            nn.Linear(num_genres, 32),
            nn.ReLU(),
            nn.Dropout(dropout)
        )

        # MLP principal
        self.fc1 = nn.Linear(embedding_dim_mlp * 2 + 32, 128)
        self.bn1 = nn.BatchNorm1d(128)
        self.fc2 = nn.Linear(128, 64)
        self.bn2 = nn.BatchNorm1d(64)
        self.dropout = nn.Dropout(dropout)

        # Capa final (concatena GMF + MLP)
        self.output = nn.Linear(embedding_dim_gmf + 64, 1)

    def forward(self, user, movie, genre):
        # GMF
        user_gmf = self.user_embedding_gmf(user)
        movie_gmf = self.movie_embedding_gmf(movie)
        gmf_output = user_gmf * movie_gmf

        # MLP
```

```python
        user_mlp = self.user_embedding_mlp(user)
        movie_mlp = self.movie_embedding_mlp(movie)
        genre_repr = self.genre_layer(genre)

        mlp_input = torch.cat([user_mlp, movie_mlp, genre_repr], dim=1)
        x = F.relu(self.bn1(self.fc1(mlp_input)))
        x = self.dropout(x)
        x = F.relu(self.bn2(self.fc2(x)))
        x = self.dropout(x)

        # Final
        final_input = torch.cat([gmf_output, x], dim=1)
        out = self.output(final_input)
        return out.squeeze()
```

loss

[20]:
```python
import torch.optim as optim

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Usando dispositivo: {device}")

model = NeuMF(
    num_users=len(user_encoder.classes_),
    num_movies=len(movie_encoder.classes_),
    num_genres=len(genre_columns),
    embedding_dim_gmf=32,
    embedding_dim_mlp=32,
    dropout=0.3
).to(device)

# Pérdida
criterion = nn.MSELoss()

# Optimizador
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)

# Scheduler: reduce el LR si no mejora la validación
scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,
    mode='min',
    factor=0.5,
    patience=2
)
```

```
Usando dispositivo: cuda
```

[25]:

```python
num_epochs = 30
best_val_loss = float('inf')
patience = 5
early_stopping_counter = 0

for epoch in range(num_epochs):
    model.train()
    total_loss = 0

    for batch in train_loader:
        users = batch['user'].to(device)
        movies = batch['movie'].to(device)
        genres = batch['genre'].to(device)
        ratings = batch['rating'].to(device)

        preds = model(users, movies, genres)
        loss = criterion(preds, ratings)

        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()

        total_loss += loss.item() * len(ratings)

    avg_train_loss = total_loss / len(train_loader.dataset)

    # VALIDACIÓN
    model.eval()
    val_loss = 0
    with torch.no_grad():
        for batch in val_loader:
            users = batch['user'].to(device)
            movies = batch['movie'].to(device)
            genres = batch['genre'].to(device)
            ratings = batch['rating'].to(device)

            preds = model(users, movies, genres)
            loss = criterion(preds, ratings)
            val_loss += loss.item() * len(ratings)

    avg_val_loss = val_loss / len(val_loader.dataset)
    scheduler.step(avg_val_loss)

    print(f"Epoch {epoch+1}/{num_epochs} | Train Loss: {avg_train_loss:.4f} | ␣
↪Val Loss: {avg_val_loss:.4f}")
```

```
    # EARLY STOPPING
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        early_stopping_counter = 0
        #torch.save(model.state_dict(), "best_model_with_genres.pth")
    else:
        early_stopping_counter += 1
        if early_stopping_counter >= patience:
            print("  Early stopping activado.")
            break
```

```
Epoch 1/30 | Train Loss: 0.0227 | Val Loss: 0.0309
Epoch 2/30 | Train Loss: 0.0224 | Val Loss: 0.0310
Epoch 3/30 | Train Loss: 0.0224 | Val Loss: 0.0311
Epoch 4/30 | Train Loss: 0.0223 | Val Loss: 0.0311
Epoch 5/30 | Train Loss: 0.0222 | Val Loss: 0.0312
Epoch 6/30 | Train Loss: 0.0221 | Val Loss: 0.0312
 Early stopping activado.
```

[26]:

```
import time
import pandas as pd
import numpy as np
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from collections import defaultdict

# Medir tiempo
start_time = time.time()

# Cargar mejor modelo
model.eval()

# Evaluación sobre el set de test
y_true, y_pred = [], []

with torch.no_grad():
    for batch in test_loader:
        users = batch['user'].to(device)
        movies = batch['movie'].to(device)
        genres = batch['genre'].to(device)
        ratings = batch['rating'].to(device)

        preds = model(users, movies, genres)

        # Desnormalizar si es necesario
        preds = preds * 5
        ratings = ratings * 5
```

```
        y_true.extend(ratings.cpu().numpy())
        y_pred.extend(preds.cpu().numpy())

y_true = np.array(y_true)
y_pred = np.array(y_pred)

rmse = np.sqrt(mean_squared_error(y_true, y_pred))
mae = mean_absolute_error(y_true, y_pred)
r2 = r2_score(y_true, y_pred)

# Precision@10 y NDCG@10
k = 10
user_preds = defaultdict(list)
user_truth = defaultdict(list)

with torch.no_grad():
    for batch in test_loader:
        users = batch['user'].to(device)
        movies = batch['movie'].to(device)
        genres = batch['genre'].to(device)
        ratings = batch['rating'].to(device)
        preds = model(users, movies, genres)

        preds = preds * 5
        ratings = ratings * 5

        for u, pred, true in zip(users.cpu().numpy(), preds.cpu().numpy(),␣
 ↪ratings.cpu().numpy()):
            user_preds[u].append(pred)
            user_truth[u].append(true)

precisions = []
for u in user_preds:
    preds_u = np.array(user_preds[u])
    truths_u = np.array(user_truth[u])
    top_k_indices = np.argsort(-preds_u)[:k]
    relevant = (truths_u >= 4.0)
    precisions.append(np.sum(relevant[top_k_indices]) / k)
precision_at_k = np.mean(precisions)

def ndcg_at_k(relevances, k):
    relevances = np.asarray(relevances)[:k]
    if relevances.size == 0:
        return 0.0
    dcg = np.sum((2 ** relevances - 1) / np.log2(np.arange(2, relevances.size +␣
 ↪2)))
```

```
    idcg = np.sum((2 ** np.sort(relevances)[::-1] - 1) / np.log2(np.arange(2,␣
↪relevances.size + 2)))
    return dcg / idcg if idcg > 0 else 0.0

ndcgs = []
for u in user_preds:
    preds_u = np.array(user_preds[u])
    truths_u = np.array(user_truth[u])
    relevances = (truths_u >= 4.0).astype(int)
    top_k_indices = np.argsort(-preds_u)[:k]
    ndcgs.append(ndcg_at_k(relevances[top_k_indices], k))
ndcg_at_k_value = np.mean(ndcgs)

# Tiempo total
total_training_time = time.time() - start_time

# Imprimir métricas
print("Métricas de Test:")
print(f"RMSE: {rmse:.4f}")
print(f"MAE : {mae:.4f}")
print(f"R²  : {r2:.4f}")
print(f"Precision@{k}: {precision_at_k:.4f}")
print(f"NDCG@{k}: {ndcg_at_k_value:.4f}")
print(f"Tiempo total de evaluación: {total_training_time:.2f} s")

# Guardar métricas
metrics = {
    "Model": "NeuMF + Genres (1M)",
    "Test RMSE": rmse,
    "Test MAE": mae,
    "Test R2": r2,
    "Precision@10": precision_at_k,
    "NDCG@10": ndcg_at_k_value,
    "Eval Time (s)": total_training_time
}

metrics_df = pd.DataFrame([metrics])
metrics_df.to_csv("neumf_1m_metrics_with_genres.csv", index=False)
print(" Métricas exportadas a 'neumf_1m_metrics_with_genres.csv'.")
```

```
Métricas de Test:
RMSE: 0.8811
MAE : 0.6898
R²  : 0.3780
Precision@10: 0.6470
NDCG@10: 0.9299
Tiempo total de evaluación: 1.87 s
 Métricas exportadas a 'neumf_1m_metrics_with_genres.csv'.
```