ratings_movies

March 28, 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.csv
ratings = pd.read_csv('../data/ml-latest-small/ratings.csv')

# Ver las primeras filas, tamaño y columnas
print(ratings.head())
print(ratings.shape)
print(ratings.columns)
```

```
   userId  movieId  rating  timestamp
0       1        1     4.0  964982703
1       1        3     4.0  964981247
2       1        6     4.0  964982224
3       1       47     5.0  964983815
4       1       50     5.0  964982931
(100836, 4)
Index(['userId', 'movieId', 'rating', 'timestamp'], dtype='object')
```

Preproccessing

[3]:
```
# Filtrar usuarios con menos de 10 ratings
user_counts = ratings['userId'].value_counts()
ratings = ratings[ratings['userId'].isin(user_counts[user_counts >= 10].index)]

# Filtrar películas con menos de 10 ratings
movie_counts = ratings['movieId'].value_counts()
ratings = ratings[ratings['movieId'].isin(movie_counts[movie_counts >= 10].
 ↪index)]

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

```
Usuarios después de filtrar: 610
Películas después de filtrar: 2269
```

[4]:
```python
# Obtener IDs únicos
unique_user_ids = ratings['userId'].unique()
unique_movie_ids = ratings['movieId'].unique()

print(f"Número de usuarios únicos: {len(unique_user_ids)}")
print(f"Número de películas únicas: {len(unique_movie_ids)}")

# Crear diccionarios de mapeo
userId_to_index = {user_id: idx for idx, user_id in enumerate(unique_user_ids)}
movieId_to_index = {movie_id: idx for idx, movie_id in␣
 ↪enumerate(unique_movie_ids)}

# Aplicar el mapeo al DataFrame
ratings['userIndex'] = ratings['userId'].map(userId_to_index)
ratings['movieIndex'] = ratings['movieId'].map(movieId_to_index)

# Comprobar
print(ratings.head())
```

```
Número de usuarios únicos: 610
Número de películas únicas: 2269
   userId  movieId  rating  timestamp  userIndex  movieIndex
0       1        1     4.0  964982703          0           0
1       1        3     4.0  964981247          0           1
2       1        6     4.0  964982224          0           2
3       1       47     5.0  964983815          0           3
4       1       50     5.0  964982931          0           4
```

[5]:
```python
# Normalizamos ratings a [0, 1]
ratings['rating_norm'] = ratings['rating'] / 5.0
print(ratings[['rating', 'rating_norm']].head())
```

```
   rating  rating_norm
0     4.0          0.8
1     4.0          0.8
2     4.0          0.8
3     5.0          1.0
4     5.0          1.0
```

Import Movies csv

[6]:
```python
# Cargar el archivo movies.csv
movies = pd.read_csv('../data/ml-latest-small/movies.csv')
print(movies.head())

# Función para obtener todos los géneros existentes
```

```python
def get_all_genres(movies_df):
    genres_set = set()
    for genres in movies_df['genres']:
        for genre in genres.split("|"):
            genres_set.add(genre)
    return list(genres_set)

all_genres = get_all_genres(movies)
genre_to_index = {genre: idx for idx, genre in enumerate(all_genres)}
num_genres = len(all_genres)

# Función para codificar los géneros en un vector one-hot
def encode_genres(genres_str, genre_to_index, num_genres):
    vec = np.zeros(num_genres, dtype=np.float32)
    for genre in genres_str.split("|"):
        if genre in genre_to_index:
            vec[genre_to_index[genre]] = 1.0
    return vec

# Crear una nueva columna con el vector de géneros
movies['genres_vector'] = movies['genres'].apply(lambda x: encode_genres(x,␣
 ↪genre_to_index, num_genres))
```

```
   movieId                               title  \
0        1                    Toy Story (1995)
1        2                      Jumanji (1995)
2        3             Grumpier Old Men (1995)
3        4            Waiting to Exhale (1995)
4        5  Father of the Bride Part II (1995)

                                        genres
0  Adventure|Animation|Children|Comedy|Fantasy
1                   Adventure|Children|Fantasy
2                               Comedy|Romance
3                         Comedy|Drama|Romance
4                                       Comedy
```

Combine Datasets

[7]:
```python
# Fusionar los datasets para agregar la información de géneros a cada rating
ratings = ratings.merge(movies[['movieId', 'genres_vector']], on='movieId',␣
 ↪how='left')
print(ratings.head())
```

```
   userId  movieId  rating  timestamp  userIndex  movieIndex  rating_norm  \
0       1        1     4.0  964982703          0           0          0.8
1       1        3     4.0  964981247          0           1          0.8
2       1        6     4.0  964982224          0           2          0.8
3       1       47     5.0  964983815          0           3          1.0
```

```
4       1       50      5.0   964982931          0            4            1.0

                                         genres_vector
0  [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, …
1  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, …
2  [0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, …
3  [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, …
4  [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, …
```

[8]:

```python
# Obtener IDs únicos y crear diccionarios de mapeo
unique_user_ids = ratings['userId'].unique()
unique_movie_ids = ratings['movieId'].unique()

print(f"Número de usuarios únicos: {len(unique_user_ids)}")
print(f"Número de películas únicas: {len(unique_movie_ids)}")

userId_to_index = {user_id: idx for idx, user_id in enumerate(unique_user_ids)}
movieId_to_index = {movie_id: idx for idx, movie_id in␣
 ↪enumerate(unique_movie_ids)}

# Aplicar el mapeo al DataFrame
ratings['userIndex'] = ratings['userId'].map(userId_to_index)
ratings['movieIndex'] = ratings['movieId'].map(movieId_to_index)
print(ratings.head())
```

```
Número de usuarios únicos: 610
Número de películas únicas: 2269
   userId  movieId  rating  timestamp  userIndex  movieIndex  rating_norm  \
0       1        1     4.0  964982703          0            0          0.8
1       1        3     4.0  964981247          0            1          0.8
2       1        6     4.0  964982224          0            2          0.8
3       1       47     5.0  964983815          0            3          1.0
4       1       50     5.0  964982931          0            4          1.0

                                         genres_vector
0  [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, …
1  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, …
2  [0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, …
3  [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, …
4  [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, …
```

[9]:

```python
ratings['rating_norm'] = ratings['rating'] / 5.0
print(ratings[['rating', 'rating_norm']].head())
```

```
   rating  rating_norm
0     4.0          0.8
1     4.0          0.8
2     4.0          0.8
3     5.0          1.0
```

```
4     5.0          1.0
```

Personalized Dataset

[10]:
```
import torch
from torch.utils.data import Dataset

class MovieLensDataset(Dataset):
    def __init__(self, users, movies, ratings, movie_features):
        self.users = users
        self.movies = movies
        self.ratings = ratings
        self.movie_features = movie_features  # vector de géneros (u otras␣
 ↪features)

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

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

Export Dataset

[11]:
```
# Asegúrate de que 'ratings' es el DataFrame que contiene la información␣
 ↪fusionada
ratings.to_csv('../data/merged_ratings_movies.csv', index=False)
print("Dataset exported successfully as 'merged_ratings_movies.csv'")
```

```
Dataset exported successfully as 'merged_ratings_movies.csv'
```

Train Test Validation Split

[14]:
```
from sklearn.model_selection import train_test_split

# 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)]

# Realizar el split por usuario
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)

print(f"Train size: {len(train_data)}")
print(f"Validation size: {len(val_data)}")
print(f"Test size: {len(test_data)}")
```

```
Train size: 56505
Validation size: 12164
Test size: 12447
```

Dataloaders

[15]:
```
from torch.utils.data import DataLoader
import numpy as np

# Convertir a tensores los índices y ratings (ya normalizados) como antes
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_norm'].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_norm'].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_norm'].values, dtype=torch.float32)

# Convertir la columna 'genres_vector' (obtenida al procesar movies.csv) en␣
↪tensores
train_movie_features = torch.tensor(np.stack(train_data['genres_vector'].
↪values), dtype=torch.float32)
val_movie_features = torch.tensor(np.stack(val_data['genres_vector'].values),␣
↪dtype=torch.float32)
test_movie_features = torch.tensor(np.stack(test_data['genres_vector'].values),␣
↪dtype=torch.float32)
```

```python
# Construcción de los datasets y dataloaders
batch_size = 512

train_dataset = MovieLensDataset(train_user, train_movie, train_rating,␣
 ↪train_movie_features)
val_dataset = MovieLensDataset(val_user, val_movie, val_rating,␣
 ↪val_movie_features)
test_dataset = MovieLensDataset(test_user, test_movie, test_rating,␣
 ↪test_movie_features)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)
test_loader = DataLoader(test_dataset, batch_size=batch_size)
```

Model

[16]:
```python
import torch.nn as nn
import torch.nn.functional as F

class NeuralCollaborativeFiltering(nn.Module):
    def __init__(self, num_users, num_movies, genre_input_dim,␣
 ↪embedding_dim=64, genre_emb_dim=32, dropout_rate=0.3):
        super(NeuralCollaborativeFiltering, self).__init__()
        self.user_embedding = nn.Embedding(num_users, embedding_dim)
        self.movie_embedding = nn.Embedding(num_movies, embedding_dim)

        # Capa para transformar el vector de géneros a un espacio de menor␣
 ↪dimensión
        self.genre_layer = nn.Linear(genre_input_dim, genre_emb_dim)

        # La entrada del MLP es la concatenación de: user_embedding,␣
 ↪movie_embedding y genre_embedding
        input_dim = embedding_dim * 2 + genre_emb_dim
        self.fc1 = nn.Linear(input_dim, 128)
        self.fc2 = nn.Linear(128, 64)
        self.output_layer = nn.Linear(64, 1)
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, user, movie, movie_features):
        user_embedded = self.user_embedding(user)
        movie_embedded = self.movie_embedding(movie)
        genre_embedded = F.relu(self.genre_layer(movie_features))

        x = torch.cat([user_embedded, movie_embedded, genre_embedded], dim=1)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
```

```
        x = self.dropout(x)
        out = self.output_layer(x)
        return out.squeeze()
```

GPU Usage

[17]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Usando dispositivo: {device}")

num_users = len(userId_to_index)
num_movies = len(movieId_to_index)
# 'num_genres' es el número de géneros únicos obtenido al procesar movies.csv
model = NeuralCollaborativeFiltering(num_users=num_users,␣
 ↪num_movies=num_movies, genre_input_dim=num_genres).to(device)
print("Modelo con integración de features de películas creado correctamente")
```

```
Usando dispositivo: cuda
Modelo con integración de features de películas creado correctamente
```

Loss Function and Optimizer

[18]:
```
import torch.optim as optim
import torch.nn as nn

criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.0005, weight_decay=1e-5)
```

Model Trainning

[19]:
```
num_epochs = 10  # Ajusta según convenga

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)
        ratings = batch['rating'].to(device)  # ya normalizado
        movie_features = batch['movie_features'].to(device)

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

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

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

    avg_train_loss = total_loss / len(train_loader.dataset)
```

```python
    # 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)
            ratings = batch['rating'].to(device)
            movie_features = batch['movie_features'].to(device)

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

    avg_val_loss = val_loss / len(val_loader.dataset)
    print(f"Epoch {epoch+1}: Train Loss = {avg_train_loss:.4f}, Val Loss =␣
↪{avg_val_loss:.4f}")
```

```
Epoch 1: Train Loss = 0.1122, Val Loss = 0.0416
Epoch 2: Train Loss = 0.0588, Val Loss = 0.0389
Epoch 3: Train Loss = 0.0522, Val Loss = 0.0365
Epoch 4: Train Loss = 0.0486, Val Loss = 0.0351
Epoch 5: Train Loss = 0.0464, Val Loss = 0.0345
Epoch 6: Train Loss = 0.0450, Val Loss = 0.0341
Epoch 7: Train Loss = 0.0435, Val Loss = 0.0332
Epoch 8: Train Loss = 0.0421, Val Loss = 0.0332
Epoch 9: Train Loss = 0.0412, Val Loss = 0.0326
Epoch 10: Train Loss = 0.0404, Val Loss = 0.0328
```

Model Storage

[ ]:
```python
torch.save(model.state_dict(), "ncf_model_current_with_movies.pth")
print("Modelo guardado correctamente.")
```

Model Evaluation

RMSE

[20]:
```python
all_preds = []
all_truth = []
model.eval()
with torch.no_grad():
    for batch in test_loader:
        users = batch['user'].to(device)
        movies = batch['movie'].to(device)
        ratings = batch['rating'].to(device)
        movie_features = batch['movie_features'].to(device)
        preds = model(users, movies, movie_features)
```

```
        # Desnormalizamos: ratings y predicciones a escala original (por␣
↪ejemplo, 0.5-5)
        all_preds.extend((preds * 5).cpu().numpy())
        all_truth.extend((ratings * 5).cpu().numpy())

all_preds = np.array(all_preds)
all_truth = np.array(all_truth)

rmse = np.sqrt(np.mean((all_preds - all_truth) ** 2))
```

MAE

[21]:
```
mae = np.mean(np.abs(all_preds - all_truth))
```

R-Square

[22]:
```
from sklearn.metrics import r2_score
r2 = r2_score(all_truth, all_preds)
```

Precision

[23]:
```
from collections import defaultdict
import numpy as np

k = 10
user_preds = defaultdict(list)
user_truth = defaultdict(list)

model.eval()
with torch.no_grad():
    for batch in test_loader:
        users = batch['user'].to(device)
        movies = batch['movie'].to(device)
        ratings = batch['rating'].to(device)
        movie_features = batch['movie_features'].to(device)
        preds = model(users, movies, movie_features)
        # Desnormalizamos: escala 0.5-5
        for u, pred, true in zip(users.cpu().numpy(), (preds * 5).cpu().
↪numpy(), (ratings * 5).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])
    # Ordenar índices según predicciones descendentes y tomar los top K
    top_k_indices = np.argsort(-preds_u)[:k]
    # Definir como relevante si el rating real es >= 4.0
```

```python
    relevant = (truths_u >= 4.0)
    num_relevant = np.sum(relevant[top_k_indices])
    precision_u = num_relevant / k
    precisions.append(precision_u)

precision_at_k = np.mean(precisions)
```

NDCG@K

[24]:
```python
def ndcg_at_k(relevances, k):
    relevances = np.asarray(relevances)[:k]
    if relevances.size == 0:
        return 0.0
    # DCG: usamos la fórmula (2^rel - 1) / log2(pos + 1)
    dcg = np.sum((2 ** relevances - 1) / np.log2(np.arange(2, relevances.size +␣
 ↪2)))
    # IDCG: DCG ideal (orden perfecto)
    ideal_relevances = np.sort(relevances)[::-1]
    idcg = np.sum((2 ** ideal_relevances - 1) / np.log2(np.arange(2,␣
 ↪ideal_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])
    # Relevancia binaria: 1 si rating >= 4.0, 0 de lo contrario
    relevances = (truths_u >= 4.0).astype(int)
    top_k_indices = np.argsort(-preds_u)[:k]
    ndcg_u = ndcg_at_k(relevances[top_k_indices], k)
    ndcgs.append(ndcg_u)

ndcg_at_k_value = np.mean(ndcgs)
```

All Metrics

[25]:
```python
from tabulate import tabulate
import pandas as pd
import numpy as np

# Suponiendo que ya tienes las variables calculadas:
# rmse, mae, r2, precision_at_k, ndcg_at_k_value

# Aseguramos que todos los valores sean Python floats y estén redondeados
metrics = {
    'RMSE': float(np.round(rmse, 4)),
    'MAE': float(np.round(mae, 4)),
    'R2': float(np.round(r2, 4)),
    'Pre': float(np.round(precision_at_k, 4)),
```

```
    'NDCG@10': float(np.round(ndcg_at_k_value, 4))
}

# Crear un DataFrame
metrics_df = pd.DataFrame(list(metrics.items()), columns=['Métrica', 'Valor'])

# Alternativa: formatear explícitamente los valores en el DataFrame
metrics_df['Valor'] = metrics_df['Valor'].apply(lambda x: f"{x:.4f}")

# Generar y mostrar la tabla con un formato más visual
tabla_formateada = tabulate(metrics_df, headers='keys', tablefmt='pretty',␣
 ↪showindex=False)
print(tabla_formateada)
```

```
+---------+--------+
| Métrica | Valor  |
+---------+--------+
|  RMSE   | 0.9151 |
|   MAE   | 0.7213 |
|   R2    | 0.2083 |
|   Pre   | 0.5048 |
| NDCG@10 | 0.8488 |
+---------+--------+
```

[26]:
```
import numpy as np
import pandas as pd
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from collections import defaultdict
import torch

# Asegúrate de que model_movies está instanciado y se ha definido el␣
 ↪dispositivo 'device'
# Cargar el checkpoint (ajusta la ruta según corresponda)
model.eval()

all_preds = []
all_truth = []

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

        preds = model(users, movies, movie_features)
```

```
        # Desnormalizamos: suponemos que se entrenó con ratings normalizados a␣
↪[0,1] → multiplicamos por 5
        preds = preds * 5
        ratings = ratings * 5

        all_preds.extend(preds.cpu().numpy())
        all_truth.extend(ratings.cpu().numpy())

all_preds = np.array(all_preds)
all_truth = np.array(all_truth)

rmse = np.sqrt(mean_squared_error(all_truth, all_preds))
mae = mean_absolute_error(all_truth, all_preds)
r2 = r2_score(all_truth, all_preds)

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)
        ratings = batch['rating'].to(device)
        movie_features = batch['movie_features'].to(device)

        preds = model(users, movies, movie_features)
        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)
    num_relevant = np.sum(relevant[top_k_indices])
    precisions.append(num_relevant / k)
precision_at_k = np.mean(precisions)

def ndcg_at_k(relevances, k):
    relevances = np.asarray(relevances)[:k]
    if relevances.size == 0:
```

```python
        return 0.0
    dcg = np.sum((2**relevances - 1) / np.log2(np.arange(2, relevances.size +␣
↪2)))
    ideal_relevances = np.sort(relevances)[::-1]
    idcg = np.sum((2**ideal_relevances - 1) / np.log2(np.arange(2,␣
↪ideal_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]
    ndcg_u = ndcg_at_k(relevances[top_k_indices], k)
    ndcgs.append(ndcg_u)
ndcg_at_k_value = np.mean(ndcgs)

# Crear el diccionario de métricas y exportar a CSV
metrics = {
    "Model": "NCFMovies (Ratings + Genres)",
    "Test RMSE": rmse,
    "Test MAE": mae,
    "Test R2": r2,
    "Precision@10": precision_at_k,
    "NDCG@10": ndcg_at_k_value
}
metrics_df = pd.DataFrame([metrics])
metrics_df.to_csv("ncf_movies_metrics.csv", index=False)
print("Model 2 metrics exported to 'ncf_movies_metrics.csv'")
```

```
Model 2 metrics exported to 'ncf_movies_metrics.csv'
```