neuMF_1m

March 28, 2025

[1]:

```python
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
```

[2]:

```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
```

[3]:
```
# 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
```

[4]:
```
from sklearn.model_selection import train_test_split

# Elegir qué rating usar
```

```
rating_col = 'rating_norm'  # cambia a 'rating_std' si quieres estandarizado

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

# 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: 697017
Validation size: 149779
Test size: 152815
```

[5]:

```
import torch
from torch.utils.data import Dataset, DataLoader

# Convertir a tensores
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)

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)

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)
```

```python
# Dataset personalizado
class MovieLensDataset(Dataset):
    def __init__(self, users, movies, ratings):
        self.users = users
        self.movies = movies
        self.ratings = ratings

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

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

# Crear datasets
train_dataset = MovieLensDataset(train_user, train_movie, train_rating)
val_dataset = MovieLensDataset(val_user, val_movie, val_rating)
test_dataset = MovieLensDataset(test_user, test_movie, test_rating)
```

[6]:
```python
from torch.utils.data import DataLoader

batch_size = 512  # Puedes ajustar esto 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
)
```

[7]:
```
import torch.nn as nn
import torch.nn.functional as F

class NeuMF(nn.Module):
    def __init__(self, num_users, num_movies, 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)

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

        # Output layer: combina GMF + MLP
        self.output = nn.Linear(embedding_dim_gmf + 64, 1)

    def forward(self, user, movie):
        # GMF path
        user_gmf = self.user_embedding_gmf(user)
        movie_gmf = self.movie_embedding_gmf(movie)
        gmf_output = user_gmf * movie_gmf  # Element-wise product

        # MLP path
        user_mlp = self.user_embedding_mlp(user)
        movie_mlp = self.movie_embedding_mlp(movie)
        mlp_input = torch.cat([user_mlp, movie_mlp], 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)

        # Combine GMF + MLP outputs
        final_input = torch.cat([gmf_output, x], dim=1)
```

```
        out = self.output(final_input)
        return out.squeeze()
```

[ ]:

[8]:
```
import torch.optim as optim
import torch.nn as nn

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_),
    embedding_dim_gmf=32,
    embedding_dim_mlp=32,
    dropout=0.3
).to(device)

# Función de pérdida
criterion = nn.MSELoss()

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

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

```
Usando dispositivo: cuda
```

[9]:
```
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)
        ratings = batch['rating'].to(device)
```

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

    # 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)

            preds = model(users, movies)
            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 (opcional)
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        early_stopping_counter = 0
        torch.save(model.state_dict(), "best_model.pth")
    else:
        early_stopping_counter += 1
        if early_stopping_counter >= patience:
            print(" Early stopping activado.")
            break
```

```
Epoch 1/30 | Train Loss: 0.0679 | Val Loss: 0.0477
Epoch 2/30 | Train Loss: 0.0427 | Val Loss: 0.0371
Epoch 3/30 | Train Loss: 0.0361 | Val Loss: 0.0346
Epoch 4/30 | Train Loss: 0.0351 | Val Loss: 0.0344
Epoch 5/30 | Train Loss: 0.0343 | Val Loss: 0.0334
Epoch 6/30 | Train Loss: 0.0336 | Val Loss: 0.0328
```

```
Epoch 7/30 | Train Loss: 0.0328 | Val Loss: 0.0322
Epoch 8/30 | Train Loss: 0.0323 | Val Loss: 0.0319
Epoch 9/30 | Train Loss: 0.0319 | Val Loss: 0.0315
Epoch 10/30 | Train Loss: 0.0315 | Val Loss: 0.0314
Epoch 11/30 | Train Loss: 0.0311 | Val Loss: 0.0313
Epoch 12/30 | Train Loss: 0.0308 | Val Loss: 0.0311
Epoch 13/30 | Train Loss: 0.0305 | Val Loss: 0.0310
Epoch 14/30 | Train Loss: 0.0303 | Val Loss: 0.0309
Epoch 15/30 | Train Loss: 0.0301 | Val Loss: 0.0309
Epoch 16/30 | Train Loss: 0.0298 | Val Loss: 0.0307
Epoch 17/30 | Train Loss: 0.0296 | Val Loss: 0.0305
Epoch 18/30 | Train Loss: 0.0294 | Val Loss: 0.0305
Epoch 19/30 | Train Loss: 0.0293 | Val Loss: 0.0305
Epoch 20/30 | Train Loss: 0.0291 | Val Loss: 0.0304
Epoch 21/30 | Train Loss: 0.0289 | Val Loss: 0.0303
Epoch 22/30 | Train Loss: 0.0288 | Val Loss: 0.0303
Epoch 23/30 | Train Loss: 0.0287 | Val Loss: 0.0304
Epoch 24/30 | Train Loss: 0.0286 | Val Loss: 0.0304
Epoch 25/30 | Train Loss: 0.0285 | Val Loss: 0.0303
Epoch 26/30 | Train Loss: 0.0284 | Val Loss: 0.0305
Epoch 27/30 | Train Loss: 0.0283 | Val Loss: 0.0303
Epoch 28/30 | Train Loss: 0.0283 | Val Loss: 0.0305
Epoch 29/30 | Train Loss: 0.0268 | Val Loss: 0.0301
Epoch 30/30 | Train Loss: 0.0264 | Val Loss: 0.0301
```

[10]:

```
import numpy as np
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Cargar el mejor modelo entrenado
model.eval()

y_true = []
y_pred = []

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

        preds = model(users, movies)

        # Si los ratings están normalizados, desnormalizamos
        preds = preds * 5
        ratings = ratings * 5

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

```python
        y_pred.extend(preds.cpu().numpy())

# Convertimos a arrays
y_true = np.array(y_true)
y_pred = np.array(y_pred)

# Cálculo de métricas
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)

print("  MÉTRICAS DE ERROR - NeuMF")
print(f"  RMSE: {rmse:.4f}")
print(f"  MAE : {mae:.4f}")
print(f"  R²  : {r2:.4f}")
```

```
 MÉTRICAS DE ERROR - NeuMF
 RMSE: 0.8667
 MAE : 0.6814
 R²  : 0.3981
```

[13]:
```python
from collections import defaultdict



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)

        preds = model(users, movies)

        # Desnormalizamos (importante para comparar con escala original)
        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)

# Precision@K
```

```
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])

    precision_u = num_relevant / k
    precisions.append(precision_u)

precision_at_k = np.mean(precisions)

# NDCG@K
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)))
    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)

print(f" Precision@{k}: {precision_at_k:.4f}")
print(f" NDCG@{k}: {ndcg_at_k_value:.4f}")
```

```
Precision@10: 0.6489
NDCG@10: 0.9317
```

[14]:

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

```python
from collections import defaultdict

# Empezamos a contar el tiempo de entrenamiento
start_time = time.time()

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

# Variables para almacenar la evolución (opcional, si quieres ver la curva)
epoch_train_losses = []
epoch_val_losses = []

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)

        preds = model(users, movies)
        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)
    epoch_train_losses.append(avg_train_loss)

    # 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)

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

```
    avg_val_loss = val_loss / len(val_loader.dataset)
    epoch_val_losses.append(avg_val_loss)
    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.pth")
    else:
        early_stopping_counter += 1
        if early_stopping_counter >= patience:
            print("Early stopping activado.")
            break

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

# Cargar el mejor modelo guardado
model.load_state_dict(torch.load("best_model.pth", map_location=device))
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)
        ratings = batch['rating'].to(device)

        preds = model(users, movies)

        # Si se entrenó con ratings normalizados, desnormalizamos a [0, 5]
        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)
```

```python
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)

# Cálculo de 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)
        ratings = batch['rating'].to(device)
        preds = model(users, movies)

        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:
        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 = []
```

```python
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)

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 entrenamiento: {total_training_time:.2f} s")

# Guardamos todas las métricas en un diccionario
metrics = {
    "Model": "NeuMF (1M)",
    "Epochs": epoch + 1,
    "Train Loss": avg_train_loss,
    "Val Loss": avg_val_loss,
    "Test RMSE": rmse,
    "Test MAE": mae,
    "Test R2": r2,
    "Precision@10": precision_at_k,
    "NDCG@10": ndcg_at_k_value,
    "Total Training Time (s)": total_training_time
}

# Convertir a DataFrame y exportar a CSV
metrics_df = pd.DataFrame([metrics])
metrics_df.to_csv("neumf_1m_metrics.csv", index=False)
print("Métricas exportadas a 'neumf_1m_metrics.csv'.")
```

```
Epoch 1/30 | Train Loss: 0.0262 | Val Loss: 0.0303
Epoch 2/30 | Train Loss: 0.0260 | Val Loss: 0.0303
Epoch 3/30 | Train Loss: 0.0248 | Val Loss: 0.0304
Epoch 4/30 | Train Loss: 0.0245 | Val Loss: 0.0306
Epoch 5/30 | Train Loss: 0.0243 | Val Loss: 0.0306
Epoch 6/30 | Train Loss: 0.0235 | Val Loss: 0.0309
Epoch 7/30 | Train Loss: 0.0232 | Val Loss: 0.0309
Early stopping activado.
Métricas de Test:
RMSE: 0.8689
MAE : 0.6830
R²  : 0.3951
Precision@10: 0.6478
```

```
NDCG@10: 0.9305
Tiempo total de entrenamiento: 32.16 s
Métricas exportadas a 'neumf_1m_metrics.csv'.
```