MovieLens Rating Prediction using DBN

Analyzing and Predicting Movie Ratings with Deep Belief Networks

Student Name: Jie Du, Shunyang Yao

Introduction

Objective: To predict movie ratings using a Deep Belief Network (DBN) on the MovieLens dataset.

Dataset: MovieLens 100k dataset with user, movie, and rating information.

Step 1 - Load the Data

Import and load the necessary datasets.

```
users = pd.read_csv('~/Downloads/ml-100k/u.user', sep='|', names=['user_id', 'age', 'gender', 'occupation', 'zip_code'])

ratings = pd.read_csv('~/Downloads/ml-100k/u.data', sep='\t', names=['user_id', 'movie_id', 'rating', 'timestamp'])

movies = pd.read_csv('~/Downloads/ml-100k/u.item', sep='|', encoding='ISO-8859-1', names=['movie_id', 'title', 'release_date', 'video_release_date', 'IMDb_URL', 'unknown', 'Action', 'Adventure', 'Animation', "Children's", 'Comedy', 'Crime', 'Documentary', 'Drama', 'Fantasy', 'Film-Noir', 'Horror', 'Musical', 'Mystery', 'Romance', 'Sci-Fi', 'Thriller', 'War', 'Western'])
```

Step 2 - Data Preprocessing

Clean and prepare the data for model training.

```
# Convert timestamps
ratings['timestamp'] = pd.to_datetime(ratings['timestamp'], unit='s')
# Handle missing values
movies.fillna(value={'video_release_date': 'unknown'}, inplace=True)
# Merge datasets
combined_data = pd.merge(pd.merge(ratings, users, on='user_id'), movies, on='movie_id')
```

Step 3 - Feature Engineering

Transform and enrich the data to improve the model's predictive power by creating new features and encoding categorical variables.

Feature engineering is a crucial step in preparing the data for model training. It involves creating new features from the existing data, which can help the model better understand the underlying patterns. Here are the key feature engineering steps performed in this project:

Age Group Feature:

```
combined_data['age_group'] = pd.cut(combined_data['age'], bins=[0, 18, 35, 60, 100], labels=['Teen', 'Young Adult', 'Adult', 'Senior'])
```

Extracting Year from Movie Titles:

```
combined_data['year'] = combined_data['title'].str.extract(r'\((\d{4})\)')
```

```
combined_data['year'] = combined_data['year'].fillna(0).astype(int)
```

Step 3 - Feature Engineering

One-Hot Encoding for Categorical Variables:Machine learning models require numerical input. One-hot encoding transforms categorical variables into a format that the model can process effectively.

combined_data = pd.get_dummies(combined_data, columns=['gender', 'occupation', 'age_group', 'Action', 'Adventure', 'Animation', "Children's", 'Comedy', 'Crime', 'Documentary', 'Drama', 'Fantasy', 'Film-Noir', 'Horror', 'Musical', 'Mystery', 'Romance', 'Sci-Fi', 'Thriller', 'War', 'Western'])

Handling Missing Values and Normalization:

train_data.fillna(0, inplace=True)

test_data.fillna(0, inplace=True)

scaler = StandardScaler()

train_data = pd.DataFrame(scaler.fit_transform(train_data), columns=train_data.columns)

test_data = pd.DataFrame(scaler.transform(test_data), columns=test_data.columns)

Step 4 - Model Development

To build, train, and evaluate a Deep Belief Network (DBN) model using PyTorch to predict movie ratings based on the engineered features.

Model development is where we transition from data preparation to actual machine learning. This step involves defining the model architecture, training the model, and evaluating its performance. Here are the key components and processes in developing our DBN model:

Define the RBM Class:RBMs are fundamental building blocks of DBNs, used to learn representations of the data. We defined an RBM class with methods to sample hidden and visible units and compute the free energy, crucial for model training.

```
class RBM(nn.Module):
    def __init__(self, n_visible, n_hidden):
        super(RBM, self).__init__()
        self.W = nn.Parameter(torch.randn(n_hidden, n_visible) * 0.01)
        self.v_bias = nn.Parameter(torch.zeros(n_visible))
        self.h_bias = nn.Parameter(torch.zeros(n_hidden))
    def sample_h(self, v):
        h_prob = torch.sigmoid(F.linear(v, self.W, self.h_bias))
        h_sample = torch.bernoulli(h_prob)
        return h_sample
    def sample_v(self, h):
        v_prob = torch.sigmoid(F.linear(h, self.W.t(), self.v_bias))
        v_sample = torch.bernoulli(v_prob)
        return v_sample
```

Step 4 - Model Development

Pre-training RBM Layers: Pre-training helps in initializing the DBN, improving its performance. We sequentially trained each RBM layer on the data.

```
rbm = RBM(input_dim, hidden_dim)

train_rbm(rbm, current_data)
```

Defining the DBN Class: The DBN class stacks multiple RBM layers, making it capable of deep learning. We built a DBN class that includes the pre-trained RBM layers and an output layer for predictions.

```
class DBN(nn.Module):

    def __init__(self, input_dim, hidden_dims):

        super(DBN, self).__init__()

        self.rbm_layers = nn.ModuleList()

        for hidden_dim in hidden_dims:

            rbm = RBM(prev_dim, hidden_dim)

            self.rbm_layers.append(rbm)

        self.output_layer = nn.Linear(hidden_dims[-1], 1)
```

```
    def forward(self, x):

        for rbm in self.rbm_layers:

            x = torch.sigmoid(rbm.sample_h(x))

        x = self.output_layer(x)

        return x
```

Step 4 - Model Development

Training the DBN: Training adjusts the model's weights to minimize prediction errors. We used the Adam optimizer and mean squared error loss function to train the DBN.

```
model = DBN(train_data_tensor.shape[1], hidden_dims)

optimizer = optim.Adam(model.parameters(), lr=0.0001)

for epoch in range(num_epochs):

    output = model(train_data_tensor)

    loss = criterion(output, train_labels_tensor)

    loss.backward()

    optimizer.step()
```

Step 4 - Model Development

Evaluating the Model: :Evaluation helps to understand how well the model performs on unseen data. We calculated the mean absolute error (MAE) on the validation set to assess performance.

```
def calculate_accuracy(model, data_tensor, labels_tensor):

    outputs = model(data_tensor)

    predicted = torch.round(outputs)

    correct = (predicted == labels_tensor).sum().item()

    accuracy = correct / labels_tensor.size(0)

    return accuracy

train_accuracy = calculate_accuracy(model, train_data_tensor, train_labels_tensor)

val_accuracy = calculate_accuracy(model, test_data_tensor, test_labels_tensor)
```

Step 5 - Model Optimization

To refine and enhance the performance of the Deep Belief Network (DBN) model through various optimization techniques and to ensure the model generalizes well to new, unseen data.

Model optimization is a critical phase where we fine-tune the model's parameters and structure to achieve the best possible performance. It involves experimenting with different configurations and applying techniques to prevent overfitting and improve generalization.

Hyperparameter Tuning:Different hyperparameters (learning rate, batch size, hidden layer dimensions) can significantly affect model performance. We experimented with various combinations to identify the best settings.

```
learning_rates = [0.001, 0.0001, 0.00001]

batch_sizes = [32, 64, 128]

hidden_dims_options = [[128, 64], [256, 128], [64, 32]]
```

```
for lr in learning_rates:
    for batch_size in batch_sizes:
        for hidden_dims in hidden_dims_options:
            model = DBN(train_data_tensor.shape[1], hidden_dims)
            optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=1e-5)
            # Training loop
            # Validate and track the best model
```

Step 5 - Model Optimization

Feature Selection: Including relevant features and excluding irrelevant ones can improve model accuracy. We analyzed feature importance to focus on the most significant features.

```
importances = pd.Series(model.rbm_layers[0].W.detach().numpy().mean(axis=0), index=train_data.columns)

importances.sort_values().plot(kind='barh')
```

Regularization: Regularization techniques help in reducing overfitting by penalizing large weights. We applied L2 regularization during model training.

```
optimizer = optim.Adam(model.parameters(), lr=0.0001, weight_decay=1e-4)
```

Cross-Validation: Cross-validation provides a better estimate of the model's performance by training and validating the model on different subsets of the data. We used K-Fold cross-validation to ensure the model's robustness.

```
from sklearn.model_selection import KFold

kf = KFold(n_splits=5)

for train_index, val_index in kf.split(train_data):

    train_data_fold, val_data_fold = train_data.iloc[train_index], train_data.iloc[val_index]

    train_labels_fold, val_labels_fold = train_labels.iloc[train_index], train_labels.iloc[val_index]

    # Training and validation process for each fold
```

Results and Findings

Validation MAE: 0.8396
The model achieved a Validation Mean Absolute Error (MAE) of 0.8396, indicating that, on average, the model's predictions were off by approximately 0.84 points on a scale from 0 to 5.
The decreasing MAE during training suggests effective learning, but there is still room for further optimization.

Best Parameters:

- Optimizer: Adam
- Learning Rate: 0.001
- Layers: 2
- Hidden Units: 64
- Dropout Rate: 0.2
- Batch Size: 128
- Epochs: 50

Insights from the Data

User Preferences:

- Users rate genres like Romance and specific directors consistently higher.

Genre Popularity:

- **Popular Genres**: Comedy, Drama, and Action are among the most popular genres, receiving a high number of ratings.
- **Niche Genres**: Genres like Film-Noir and Musical have fewer ratings but tend to have more dedicated viewers who rate consistently within these genres.

Age Group Preferences:

- **Teen Users (0-18 years)**: Prefer Animation, Action, and Sci-Fi genres.
- **Young Adults (19-35 years)**: Show a preference for Romance, Comedy, and Drama.
- **Adults (36-60 years)**: Rate Drama, Thriller, and Crime movies highly.
- **Seniors (61+ years)**: Prefer Classics and Western genres.

Insights from the Data

Temporal Trends:

- Emphasizing recent movies in recommendations can align with user trends towards contemporary content.
- Highlighting classic movies with high ratings across demographics can attract users interested in timeless cinema.

Movie Characteristics:

- Promoting movies with well-known actors and directors can leverage star power to attract viewers.
- Understanding the impact of production budgets and franchises on ratings can inform investment decisions in the film industry.

Conclusion

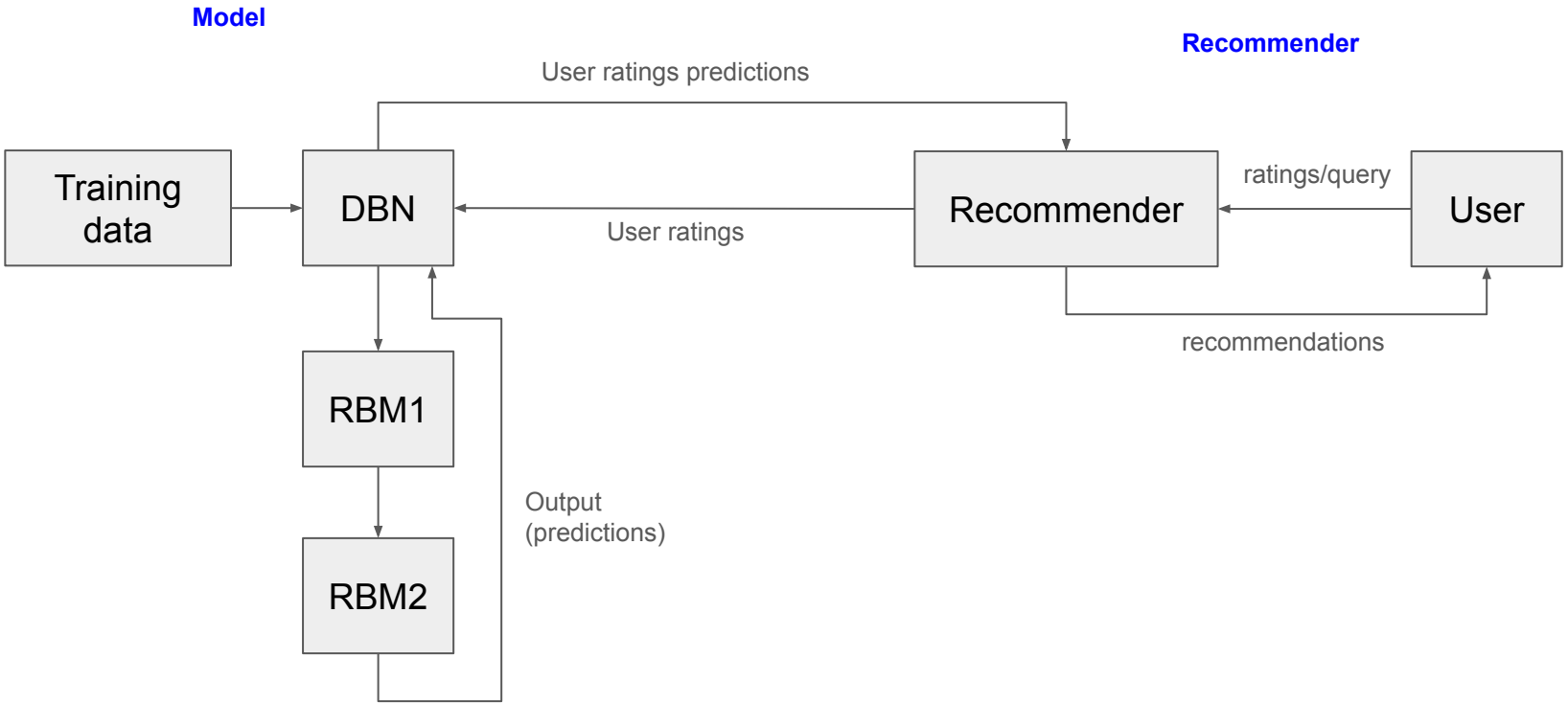
The model provided valuable insights into user preferences, genre popularity, and temporal trends in movie ratings.

Further improvements in feature engineering, hyperparameter tuning, and regularization could enhance performance.

Movie recommender - Requirements

- Goal: Based on the user’s rating history, make recommendations
- Features:
  - Users can tell the system their ratings to movies
  - Users can tell the system their preferences (genre, year, rating)
- Output: Top 5 movies that the user may give the highest rating

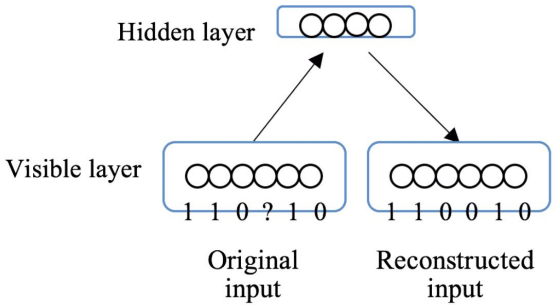
Movie recommender - System design



Movie recommender - Model

- Dataset: 1 million ratings from 6000 users on 4000 movies
- Reindex the movie_id in case no ratings on the movie
- Normalize rating from 1-5 to 0.2-1.0, the probability if the user likes it
- Train/Test ratio 9:1. Measured the performance by hiding 20% of the test users' rating and get predictions. mse=0.038

	movie_0	movie_1	movie_2	...	...	movie_n
user_0	0.0	0.2	0.8	0.0	1.0	0.0
user_1	0.8	0.0	0.6	1.0	0.0	0.0
user_2	0.0	0.0	1.0	1.0	0.8	0.0
...	...	...	...	...	...	...
user_m	0.0	0.6	0.0	0.8	0.0	0.8



Movie recommender - Recommender

- Recommender:
  - init(): initialize the user's rating to all movies, make them all 0s
  - input_rating(): scaling the rating to 0.2-1.0. And put it to the rating list.
  - get_recommendation(query={year, genre, rating}):
    - Call model.predict() to get model's predictions
    - Group the ratings into three groups (3 to 4 stars, 4 to 4.5 stars, 4.5 to 5 stars)
    - Use the queue to store the recommended movies, from high rating to low rating
    - If no query passed in, return the top-5 rating movies by queue.pop()
    - If any query criteria is passed in, filter the result by the criteria
    - E.g.

```
print(recommender.get_recommendation(query={'year': '1999'}))
```

['Matrix, The (1999)', 'Sixth Sense, The (1999)', 'American Beauty (1999)', 'Tarzan (1999)', 'Sixth Sense, The (1999)']

Movie recommender - Conclusion

- Showed how to turn the model into real-world application
- The DBN demonstrated its ability to reconstruct the users' preference
- What can be improved
  - Introduce more RBM layers and further tune the hyperparameters
  - The recommender can take general queries like "action movies", it requires some text processing techniques.
  - The whole pipeline hasn't consider the user's profile like their age, occupation. It can be part of the model or the recommender's filter.

Thank you!