# README: Autoencoder-Based Recommendation System

## Overview

This project implements an autoencoder-based recommendation system for suggesting relevant items (e.g., categories or investments) to users. The recommendation system evaluates user interactions with items and optimizes performance using metrics such as Precision@K, Recall@K, and NDCG@K.

## Features

- Preprocesses user-item interaction data with additional features (e.g., `spended\_time`, `amount`).  
- Uses an autoencoder model for collaborative filtering.  
- Implements evaluation metrics:  
 - \*\*Precision@K\*\*  
 - \*\*Recall@K\*\*  
 - \*\*NDCG@K\*\*  
- Tracks and saves the model with the best Precision@K during training.

## Requirements

### Dependencies

- Python 3.8+  
- Libraries:  
 - torch  
 - numpy  
 - pandas  
 - scikit-learn  
  
Install dependencies using:  
```bash  
pip install torch numpy pandas scikit-learn  
```

### Data

The input data file must be a CSV named `user\_investment\_data\_v2.csv` with the following columns:  
- `userid`: Unique user IDs.  
- `category`: Category names or labels.  
- `spended\_time`: Time spent by the user in a category.  
- `amount`: Amount invested by the user in a category.  
  
Ensure the dataset is correctly formatted before running the code.

## Files

- \*\*model.py\*\*: Contains the `Autoencoder` model and the `DataPreprocessor` class for data preparation.  
- \*\*train.py\*\*: Main script for training and evaluating the recommendation system.

## Usage

### Training the Model

1. Ensure `user\_investment\_data\_v2.csv` is in the same directory.  
2. Run the training script:  
 ```bash  
 python train.py  
 ```  
3. The script trains the autoencoder for 1000 epochs by default, evaluates performance every 50 epochs, and saves the best-performing model based on `Precision@K` to `best\_model.pth`.

### Evaluation Metrics

The model uses the following metrics to assess performance:  
- \*\*Precision@K\*\*: Measures how many of the top-K recommended items are relevant.  
- \*\*Recall@K\*\*: Measures how many relevant items are in the top-K recommendations.  
- \*\*NDCG@K\*\*: Evaluates ranking quality by considering the position of relevant items in the top-K recommendations.  
  
Example metrics output during evaluation:  
```  
Evaluation Metrics: Loss=0.1234, Precision@5=0.8765, Recall@5=0.7890, NDCG@5=0.8234  
```

### Saved Model

- The best-performing model is saved as `best\_model.pth`.  
- Load the saved model for inference or further evaluation using:  
 ```python  
 model = Autoencoder(input\_dim, hidden\_dim)  
 model.load\_state\_dict(torch.load("best\_model.pth"))  
 model.eval()  
 ```

## Code Structure

### model.py

#### Autoencoder Class  
Defines a simple autoencoder with one hidden layer:  
```python  
class Autoencoder(nn.Module):  
 def \_\_init\_\_(self, input\_dim, hidden\_dim):  
 super(Autoencoder, self).\_\_init\_\_()  
 self.encoder = nn.Sequential(  
 nn.Linear(input\_dim, hidden\_dim),  
 nn.ReLU(True)  
 )  
 self.decoder = nn.Sequential(  
 nn.Linear(hidden\_dim, input\_dim),  
 nn.Sigmoid()  
 )  
  
 def forward(self, x):  
 encoded = self.encoder(x)  
 decoded = self.decoder(encoded)  
 return decoded  
```

#### DataPreprocessor Class  
Prepares user-item interaction data:  
- Encodes categories.  
- Normalizes `spended\_time` and `amount`.  
- Creates a user-item interaction matrix.

### train.py

Contains the main training loop and evaluation logic:  
- Trains the autoencoder.  
- Computes recommendation metrics (`Precision@K`, `Recall@K`, `NDCG@K`).  
- Saves the model with the best Precision@K.

## Customization

- \*\*Modify Hidden Layer Size:\*\* Change `hidden\_dim` in the `train\_model` function for a different latent representation.  
- \*\*Adjust Metrics:\*\* Change the value of `k` in `evaluate\_model` to compute metrics for different recommendation list sizes.  
- \*\*Extend Features:\*\* Include additional user or item features by updating the `DataPreprocessor` class.

## Future Improvements

- Incorporate side information (e.g., user demographics).  
- Use advanced autoencoder architectures (e.g., denoising autoencoders).  
- Implement hyperparameter optimization for better performance.