

# Recommendation System Guide/Primer

Austin Lee

February 13, 2024

# 1 Introduction

Recommendation systems are essential tools that help individuals discover content tailored to their preferences, enhancing the personalization of digital experiences within platforms and organizations. Whether suggesting movies, music, or products, these systems analyze user interactions and feedback to provide personalized recommendations, creating a more engaging and user-centric environment.

Imagine a scenario where a company curates a collection of preferences of movie watchers and various attributes associated with each movie. These user preferences may be revealed explicitly through actions like ratings, likes, impressions, and the time spent engaging with items. Movie attributes may be based on certain qualitative factors such as genre, release date, and movie time.

Building a recommendation system involves deciphering user preferences and finding commonalities among them. If two users share similar tastes in movies, the system leverages these patterns to suggest films that align with their interests. This process transforms raw data into actionable insights, enabling the platform to deliver tailored recommendations.

To illustrate a concrete one-dimensional example, say there are five customers that have given ratings to five movies, in which two of these customers have similar ratings for the first four movies.

	Movie 1 Rating	Movie 2 Rating	Movie 3 Rating	Movie 4 Rating	Movie 5 Rating
Customer One	1	3	3	3	5
Customer Two	2	1	1	2	?
Customer Three	2	3	3	3	?
Customer Four	3	4	4	1	?
Customer Five	5	1	2	2	?

Figure 1: Simple One Dimensional Example

Given customer one and customer three's revealed preferences illustrated through ratings, it is likely that customer three would enjoy movie 5. This is a simplified example for illustration purposes, however, generally recommendation systems run based on a multitude of dimensions/features. Dimensionality is typically dependent on the following:

- **Model Capacity:** more dimensionality may assist in capturing patterns and relationships within the data
- **Size of data set:** larger data sets may allow for a variety of patterns to present and for the model to better adapt

- **Memory footprint:** certain dimensions consume less/more memory and as such, in a resource-constrained environment, less dimensions may be more feasible

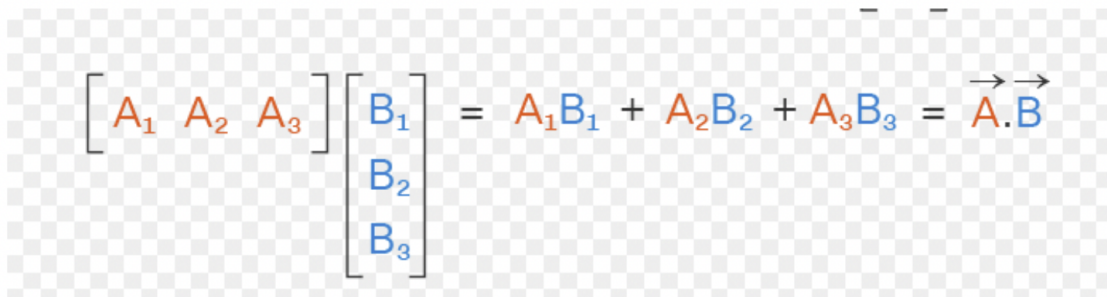
The following section details a specific example of how to implement a recommendation system. This document serves as my own primer/guide to creating a recommendation system.

Credit to the following for assisting in my own understanding of this process:

- Serrano.Academy
- Professor Yannet at University of San Francisco

## 2 Matrix Factorization

Matrix factorization is a fundamental technique used in recommendation system models to predict embeddings (think of an assigned quantitative element for features). In the context of linear algebra, matrix factorization involves breaking down a matrix into the product of two lower-dimensional matrices. These lower-dimensional matrices represent features, such as recency preference, favorite genres, or movie time preference. The computation involves a dot product of these matrices to generate an output, contributing to the prediction process in recommendation systems. The below is a refresher on the computation of dot product.



$$\begin{bmatrix} A_1 & A_2 & A_3 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} = A_1B_1 + A_2B_2 + A_3B_3 = \vec{A} \cdot \vec{B}$$

Figure 2: Dot Product

Say we have one user and one movie that we would like to assign a predicted rating. In this example, the user has two known embeddings that are as follows:

- $[0.4, -0.3]$  where 0.4 represents the user's preference for action/drama movies and  $-0.3$  represents the user's preference for new/old movies.
- $[-.1, .2]$  where  $-.1$  is an embedding that represents whether the movie leans towards action/drama and  $.2$  is an embedding a representation of the movie's release year.

By taking the dot product, ie  $.4 * -.1 + -.3 * .2$ , we arrive at the predicted rating of  $-0.1$ . This dot product captures the interaction between the user's embedded preferences and the movie's dimensional embeddings. This dot product encapsulates the interplay between the user's embedded preferences and the movie's dimensional embeddings. A higher dot product suggests stronger user interest, while a lower one indicates a lower predicted rating.

The following diagram illustrates the process of matrix factorization applied to user and item preferences.

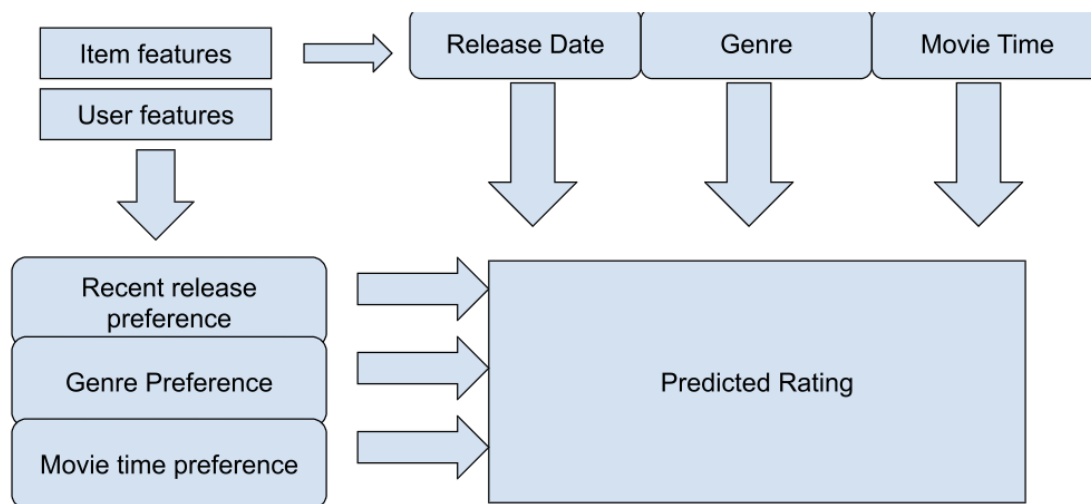


Figure 3: Item Features by User Features Factorization

To learn more about matrix factorization, here are additional resources to read:

- Matrix Factorization Explained

In the next section, we will dive into an example of matrix factorization and walk through the implementation of a machine learning model to better understand how it operates in a real-world scenario.

### 3 Matrix Factorization Example 1

This example that I will run was created by Associate Professor Yannet at University of San Diego. Please refer to her page, linked here for additional information. This document serves as my own primer/guide to creating a recommendation system and may assist in simplifying information within the notebook.

In this example, we have a dataset with the fields, userid, movieid, and rating. The dataset that we may is downloadable here.

Download the following libraries and download the following data:

```

from pathlib import Path
import pandas as pd
import numpy as np
import torch # apparently pytorch doesn't download
#with most recent python verison - need to downgrade interpreter
import torch.nn as nn
import torch.nn.functional as F

```

```

data = pd.read_csv("/Users/austinblee/Downloads/ml-25m/ratings.csv")
data.head()
print(data.head())

#sets up the data so that 80% of the data is used for training and the remaining 20%
#is for validation. Random seed 3 is used for reproducibility
np.random.seed(3)
msk = np.random.rand(len(data)) < 0.8
train = data[msk].copy()
val = data[~msk].copy()

```

### 3.1 Encoding

Although the user id and movie ids are already based in numerical integer format, encoding is used for userid and movieid to create for the purposes of contiguous indexing. Collaborative filtering models, recommendation systems, typically use matrix factorization techniques that require indexing users and items, movies in this case, using contiguous integers. This is important for creating embedding layers or matrices with consistent dimensions. More specifically having contiguous indexing allows for:

- **Compatibility with Libraries:** Some machine learning libraries and frameworks are optimized for arrays with contiguous indices. Using contiguous indices can enhance compatibility and performance with these libraries.
- **Memory Efficiency:** In many programming languages, array or matrix operations with contiguous indices can be more memory-efficient and faster compared to non-contiguous indices.
- **Model Consistency:** Having contiguous indices ensures that the model's internal representation (embedding layers, weight matrices) is consistent. It avoids potential complications arising from non-contiguous or sparse indexing.

The following functions are used for encoding the userids/movieids.

I've included comments that detail how the function is executed.

```

#proc_col encodes pandas column with continuous ids -> [10,20,30] ->
# {}
def proc_col(col, train_col=None): #train_col for training dataset
    """Encodes a pandas column with continuous ids.
    """
    if train_col is not None:
        uniq = train_col.unique()
    else:
        uniq = col.unique()
    # iterates over the pairs i,o and returns elements
    #from o and corresponding indices from i
    name2idx = {o:i for i,o in enumerate(uniq)}
    #ex: dictionary for uniq [10,20,30] would return {10:0,20:1,30:2}
    return name2idx, np.array([name2idx.get(x, -1) for x in col]), len(uniq)

```

```

# returns dictionary of elements and indices
# returns encoded index values on the initial column
# returns number of unique ids
def encode_data(df, train=None):
    """ Encodes rating data with continuous user and movie ids.
    If train is provided, encodes df with the same encoding as train.
    """
    df = df.copy()
    for col_name in ["userId", "movieId"]:
        train_col = None
        if train is not None:
            train_col = train[col_name]
    #converts unique userids or movieids into continuous integers using proc_col
    #function and stores in _,col,_
    #(gets the second return value of the proc_col function)
    _,col,_ = proc_col(df[col_name], train_col)
    df[col_name] = col
    #replaces the original column with the encoded column
    df = df[df[col_name] >= 0]
    return df

```

This is the result of the encoding function:

	userId	movieId	rating	timestamp
0	1	296	5.0	1147880044
1	1	306	3.5	1147868817
2	1	307	5.0	1147868828
3	1	665	5.0	1147878820
6	1	1175	3.5	1147868826
	userId	movieId	rating	timestamp
0	0	0	5.0	1147880044
1	0	1	3.5	1147868817
2	0	2	5.0	1147868828
3	0	3	5.0	1147878820
6	0	4	3.5	1147868826

Figure 4: Encoding Example

In the next section, we are going to create a matrix factorization class. The matrix factorization

class will generate random embeddings based on given users and items. The embedding function is essentially a lookup table that is initialized with random values and updated during training to minimize the difference between the predicted scores and the actual target values. Generally, embeddings in a matrix factorization model are learned during the training process through backpropagation.

The forward function ultimately calculates the element-wise multiplication of user and item embeddings, which results in a matrix. The 'sum(1)' adds up the said matrix resulting in a vector of predicted ratings. Said simply, this is the dot product.

```
class MF(nn.Module):
    def __init__(self, num_users, num_items, emb_size=100):
        super(MF, self).__init__()
        self.user_emb = nn.Embedding(num_users, emb_size)
        self.item_emb = nn.Embedding(num_items, emb_size)
        self.user_emb.weight.data.uniform_(0, 0.05)
        self.item_emb.weight.data.uniform_(0, 0.05)

    def forward(self, u, v):
        u = self.user_emb(u)
        v = self.item_emb(v)
        return (u * v).sum(1)
```

The following is an example of this class instance:

```
# Example: Consider 3 users, 4 items, and an embedding size of 2
num_users = 3
num_items = 4
emb_size = 2
# Create an instance of the matrix factorization model
model = nn.Embedding(num_users, emb_size), nn.Embedding(num_items, emb_size)
# Assume training data: user 1 interacts with item 2 with a rating of 4.0
user_index = torch.tensor([1])
item_index = torch.tensor([2])
rating = torch.tensor([4.0])
# Forward pass: calculate the predicted rating
user_embedding = model[0](user_index) # Embedding for user 1
item_embedding = model[1](item_index) # Embedding for item 2
predicted_rating = (user_embedding * item_embedding).sum(1)
# Print the results
print("User Embedding Matrix:")
print(user_embedding)
print("\nItem Embedding Matrix:")
print(item_embedding)
print("\nElement-wise Multiplication Matrix:")
print(user_embedding * item_embedding)
```

```
print("\nPredicted Rating:")
print(predicted_rating)
```

Output: User Embedding Matrix:

```
tensor([[0.3318, -0.0894]], grad_fn=<EmbeddingBackward>)

Item Embedding Matrix:
tensor([[0.5754, -0.2370]], grad_fn=<EmbeddingBackward>)

Element-wise Multiplication Matrix:
tensor([[0.1907, 0.0212]], grad_fn=<MulBackward0>)

Predicted Rating:
tensor([0.2119], grad_fn=<SumBackward1>)
```

Figure 5: Example Output

Here we can see that a predicted rating calculated on two randomly generated user embeddings and randomly generated movie embeddings. By taking the dot product of these two matrices, we arrive at the predicted rating.

The following section details the training of the model.

### 3.2 Training MF Model

```
def train_and_eval(model, epochs=10, lr=0.01, wd=0.0):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=wd)
    model.train()
    for i in range(epochs):
        users = torch.LongTensor(df_train.userId.values)
        items = torch.LongTensor(df_train.movieId.values)
        ratings = torch.FloatTensor(df_train.rating.values)
        y_hat = model(users, items)
        loss = F.mse_loss(y_hat, ratings)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        print(loss.item())
    model.eval()
    users = torch.LongTensor(df_val.userId.values)
    items = torch.LongTensor(df_val.movieId.values)
    ratings = torch.FloatTensor(df_val.rating.values)
    y_hat = model(users, items)
    loss = F.mse_loss(y_hat, ratings)
    print("test loss %.3f " % loss.item())
```



**Explanation for function:** Initializes an optimizer to update the model parameters during training. It uses the mean squared error (MSE) loss as the objective function Set the model to training mode. This is important for models that have different behavior during training and evaluation

For context, an epoch is one complete pass through the entire training dataset during the training of a model. During each epoch, the model's parameters are updated based on the gradients of the loss function with respect to those parameters. The model goes through 10 iterations of training. The following is a simplified summary of the function:

- **Training Epoch:** An epoch consists of iterating through all batches of the training dataset once.
- **Updating Parameters:** The model makes predictions, calculates the loss (a measure of how well the model is performing), computes gradients (think of whether its directionally correct) with respect to the parameters, and updates the parameters using an optimization algorithm.

`torch.LongTensor(dftrain.userId.values)`- converts the column userid into a PyTorch LongTensor. This tensor contains the user IDs from the training dataset, and it's now in a format that can be used as input to a PyTorch model for training or evaluation. Similarly FloatTensor converts the column ratings into a float tensor, which can be used as input for training.

Continuation of explanation:

- **Forward Pass:** What automatically calls the forward function from the MF class and computes the predicted rating using matrix factorization. Compute Loss: You compute the loss by comparing the model predictions to the actual target values.
- **Backward Pass:** Gradients are values calculated during the backward pass of the optimization process, indicating the direction and magnitude of the change needed to minimize the loss. Think of gradient as a direction to move the embedding towards.
  - `optimizer.zero_grad()`: Clears the gradients of all optimized tensors.
  - You call `loss.backward()` to compute the gradients of the loss with respect to the model parameters (backpropagation).
- **OptimizerStep:** You call `optimizer.step()`. The optimizer takes care of updating the model parameters using the computed gradients.

See the results below:

```
train_and_eval(model, epochs=10, lr=0.1)
```

```
12.910295486450195
4.848785400390625
2.6106131076812744
3.09657621383667
0.8506607413291931
1.8248921632766724
```

```
2.6612296104431152
2.142143726348877
1.0970792770385742
0.9773465394973755
test loss 1.850
```

With some fine tuning - embed = 85 and epoch = 30

```
12.909828186035156
4.846950531005859
2.61000657081604
3.091350793838501
0.8498722910881042
1.8239704370498657
2.657252073287964
2.1360974311828613
1.0926792621612549
0.9784851670265198
1.64142906665802
1.352852702140808
0.7794235348701477
0.9205158352851868
1.2968428134918213
1.2914644479751587
0.9259457588195801
0.6771771311759949
0.8546032309532166
1.0208683013916016
0.8078588247299194
0.6246208548545837
0.7215652465820312
0.8439648151397705
0.7729355692863464
0.6057626605033875
0.5709357261657715
0.6594182848930359
0.6394765973091125
0.5213302373886108
test loss 0.804
```

As you can see, through the progression of training the MSE is minimized and the model improves. When we evaluate the model against the validation dataset, we can see that loss is minimized to .8. The mse is calculated as the average of squared differences between predicted and actual values. This means that on average the model's prediction deviates .8 from its true rating.

## 4 Matrix Factorization Example 2

In the next example, I wanted to demonstrate that not all embeddings necessarily need to be randomized. That is, if we have certain known features, we could incorporate them into the

model as well. I set a couple of embeddings constant, and did not allow the model to train over these embeddings. This is largely done for illustrative purposes,

```
class MF_with_known_feature(nn.Module):
    def __init__(self, num_users, num_items, emb_size=100, known_feature_size=1):
        super(MF_with_known_feature, self).__init__()
        # Embeddings with learnable parameters
        self.user_embedding_learnable = nn.Embedding(num_users + 1, emb_size)
        self.item_embedding_learnable = nn.Embedding(num_items + 1, emb_size)
        self.user_embedding_known_feature = nn.Embedding(num_users + 1, known_feature_size)
        # Initialize learnable embeddings with uniform random values
        self.user_embedding_learnable.weight.data.uniform_(0, 0.05)
        self.item_embedding_learnable.weight.data.uniform_(0, 0.05)

    def forward(self, u, v):
        u_learnable = self.user_embedding_learnable(u)
        v_learnable = self.item_embedding_learnable(v)
        # Retrieve the known feature embedding for the user
        known_feature = self.user_embedding_known_feature(u)
        # Combine the known feature with the learnable user embedding
        u_combined = u_learnable + known_feature
        return (u_combined * v_learnable).sum(1)

data = pd.DataFrame({
    'userId': [1, 2, 3, 1, 2, 3],
    'movieId': [101, 102, 103, 101, 102, 103],
    'rating': [5.0, 4.0, 3.5, 4.5, 3.0, 2.5],
    'age': [25, 30, 22, 25, 30, 22] # Known feature (e.g., user age)
})

# Split the data into training and validation sets
msk = np.random.rand(len(data)) < 0.8
train = data[msk].copy()
val = data[~msk].copy()
# Encode the datasets
df_train = encode_data(train)
df_val = encode_data(val, train)
num_users = data['userId'].nunique()
num_items = data['movieId'].nunique()
# Create an instance of the model
model = MF_with_known_feature(num_users, num_items, emb_size=100, known_feature_size=1)
# Train the model
train_epocs(model, epochs=30, lr=0.1)

Epoch 1, Loss: 18.377811431884766
Epoch 2, Loss: 17.85274314880371
Epoch 3, Loss: 7.614724159240723
```

Epoch 4, Loss: 3.347332715988159  
Epoch 5, Loss: 4.5091047286987305  
Epoch 6, Loss: 5.372431755065918  
Epoch 7, Loss: 5.4306135177612305  
Epoch 8, Loss: 6.035780429840088  
Epoch 9, Loss: 6.029953956604004  
Epoch 10, Loss: 5.014727592468262  
Epoch 11, Loss: 4.319084167480469  
Epoch 12, Loss: 3.6510519981384277  
Epoch 13, Loss: 2.7876508235931396  
Epoch 14, Loss: 2.5966548919677734  
Epoch 15, Loss: 3.2326247692108154  
Epoch 16, Loss: 2.9745686054229736  
Epoch 17, Loss: 1.4274907112121582  
Epoch 18, Loss: 0.3394986689090729  
Epoch 19, Loss: 0.58881676197052  
Epoch 20, Loss: 1.3810782432556152  
Epoch 21, Loss: 1.770911455154419  
Epoch 22, Loss: 1.5363843441009521  
Epoch 23, Loss: 1.0725982189178467  
Epoch 24, Loss: 0.7443211078643799  
Epoch 25, Loss: 0.6081923842430115  
Epoch 26, Loss: 0.6119824647903442  
Epoch 27, Loss: 0.6317645907402039  
Epoch 28, Loss: 0.6213995218276978  
Epoch 29, Loss: 0.6336581110954285  
Epoch 30, Loss: 0.7048205137252808

Both the learnable and known features (age) are initialized randomly and updated during training to capture patterns. However, with the known feature, age, the model may learn to associate certain patterns w.r.t age to make predictions. The optimization process will adjust itself during training and capture the contribution of age within the prediction.

In the context of recommendation systems, including known features such as:

- **Explicit Interactions:** This refers to user-item interactions like ratings, clicks, or purchases. The embeddings for users and items are learned from these interactions.
- **Additional Features:** These can include demographic information, user preferences for specific genres (as in your example of action movies), or any other relevant information. These features are used to enhance the embeddings and provide a more personalized recommendation.

May be beneficial for the following reasons:

- **Improved Personalization:** Known features can capture specific aspects of user preferences that may not be evident from their explicit interactions alone.
- **Interpretability:** If certain features are known to strongly influence user preferences, incorporating them explicitly into the model allows for more interpretable recommendations.

- **Domain Expertise:** Leveraging domain-specific knowledge by including known features can enhance the model's performance.

In summary, we've covered how recommendation systems work, using matrix factorization. We saw that tweaking known features, like age, may help a model get better at predicting what you might like.