In [1]:

```
# RFM/Matrix Recommender Engine
#
# Igor Mol <igor.mol@makes.ai>
#
# A recommender system within the RFM (Recency, Frequency, Monetary) model i
# implemented using the cosine similarity metric to measure the similarity
# between users based on their purchase behavior. The RFM model focuses on
# assessing customers' recency of purchases, the frequency of transactions,
# the monetary value of their spending. By representing users as vectors in
# multidimensional space, where each dimension corresponds to one of these R
# metrics, cosine similarity calculates the cosine of the angle between thes
# vectors. The resulting similarity scores indicate how closely the purchasi
# patterns of different users align. In the recommender system, higher cosir
# similarity values imply greater similarity between users, facilitating the
# identification of customers with comparable preferences. This information
# then leveraged to generate personalized recommendations for users, enhanci
# the system's ability to suggest items that align with their historical
# purchasing behavior.

import pandas as pd
import numpy as np

# load_data():
# This function loads data from a CSV file, processes the 'purchase_date' co
# as a datetime object, and returns the resulting DataFrame.
#
# Parameters:
# - file_path: The file path to the CSV file containing the data.
#
# Returns:
# A pandas DataFrame containing the loaded data, with the 'purchase_date'
# column converted to datetime format.

def load_data(file_path):
    # Read data from the specified CSV file into a DataFrame
    data = pd.read_csv(file_path)

    # Convert the 'purchase_date' column to datetime format
    data['purchase_date'] = pd.to_datetime(data['purchase_date'])

    # Return the processed DataFrame
    return data

# calculate_rfm_scores():
# This function calculates RFM (Recency, Frequency, Monetary) scores for eac
# customer based on their purchase history. It assigns scores to customers b
# on how recent their last purchase was (Recency), how frequently they make
# purchases (Frequency), and how much money they spend (Monetary).
#
# Parameters:
# - data: DataFrame containing purchase data with columns 'customer_id',
#   'purchase_date', 'item_id', and 'amount'.
#
# Returns:
# A DataFrame with columns 'customer_id' and 'rfm_score', representing the
# calculated RFM scores for each customer.
```

```
def calculate_rfm_scores(data):
    # Find the current date as the maximum purchase date in the dataset
    current_date = data['purchase_date'].max()

    # Group the data by customer_id and aggregate metrics for recency, frequ
    # and monetary
    rfm_data = data.groupby('customer_id').agg({
        'purchase_date': lambda x: (current_date - x.max()).days,
        'item_id': 'count',
        'amount': 'sum'
    }).reset_index()

    # Calculate relative scores for recency, frequency, and monetary metrics
    rfm_data['recency_score'] = rfm_data['purchase_date'] / rfm_data['purcha
    rfm_data['frequency_score'] = rfm_data['item_id'] / rfm_data['item_id'].
    rfm_data['monetary_score'] = rfm_data['amount'] / rfm_data['amount'].max

    # Combine the scores to calculate the overall RFM score for each custome
    rfm_data['rfm_score'] = rfm_data['recency_score'] + rfm_data['frequency_

    # Return a DataFrame with customer_id and rfm_score columns
    return rfm_data[['customer_id', 'rfm_score']]

def create_user_item_matrix(data):
    return pd.pivot_table(data, values='amount', index='customer_id', columr

# Cosine similarity:
# This function computes the cosine similarity matrix from scratch for a giv
# Cosine similarity measures the cosine of the angle between two non-zero ve
# is often used to assess the similarity between rows of a matrix.
#
# Parameters:
# - matrix: The input matrix for which cosine similarity is to be computed.
#
# Returns:
# A pandas DataFrame representing the cosine similarity matrix, where each e
# represents the cosine similarity between row i and row j of the input matr

def compute_cosine_similarity_from_scratch(matrix):
    # Normalize the input matrix by dividing each row by its Euclidean norm
    matrix_normalized = matrix.div(np.linalg.norm(matrix, axis=1), axis=0)

    # Calculate the cosine similarity matrix by taking the dot product of th
    similarity_matrix = matrix_normalized @ matrix_normalized.T

    # Return the resulting similarity matrix as a DataFrame with row and col
    return pd.DataFrame(similarity_matrix, index=matrix.index, columns=matri


# get_recommendations():
# This function provides personalized recommendations for a given user based
# collaborative filtering. It calculates recommendations by considering the
# past purchases, the similarity between the user and other users, and addit
# information from RFM (Recency, Frequency, Monetary) data.
#
# Parameters:
# - user_id: The identifier of the target user for whom recommendations are
# - similarity_matrix: A matrix representing the similarity between users.
# - user_item_matrix: A matrix representing the user-item interactions (purc
#   rfm_data: Additional data containing RFM scores for customers
```

```
# - rfm_data: Additional data containing RFM scores for customers.
# - n: The number of recommendations to return (default is 5).
#
# Returns:
# A list of tuples containing recommended items for the user, where each tup
# consists of (item_id, recommendation_score).

def get_recommendations(user_id, similarity_matrix, user_item_matrix, rfm_da
    # Extract the user's past purchases from the user-item matrix
    user_purchases = user_item_matrix.loc[user_id]

    # Get a list of similar users, sorted by similarity in descending order
    similar_users = similarity_matrix[user_id].sort_values(ascending=False).

    # Initialize an empty list to store recommendations
    recommendations = []

    # Iterate over all items in the user-item matrix
    for item_id in user_item_matrix.columns:
        # Check if the user has not purchased the item
        if user_purchases[item_id] == 0:
            # Calculate the weighted sum of similar users' purchases for the
            weighted_sum = sum(
                user_item_matrix.loc[similar_user, item_id] *
                similarity_matrix.loc[user_id, similar_user] *
                rfm_data[rfm_data['customer_id'] == similar_user]['rfm_score
                for similar_user in similar_users
            )
            # Append the item and its weighted sum to the recommendations li
            recommendations.append((item_id, weighted_sum))

    # Sort recommendations by the calculated scores in descending order and
    # the top 'n'
    recommendations = sorted(recommendations, key=lambda x: x[1], reverse=Tr

    # Return the final list of recommendations
    return recommendations


# print_recommendations():
# This function generates and prints personalized recommendations for each u
# in the user-item matrix based on collaborative filtering. It utilizes the
# get_recommendations function to calculate recommendations for each user.
#
# Parameters:
# - user_item_matrix: A matrix representing the user-item interactions (purc
# - similarity_matrix: A matrix representing the similarity between users.
# - rfm_data: Additional data containing RFM scores for customers.

def print_recommendations(user_item_matrix, similarity_matrix, rfm_data):
    # Iterate over each user in the user-item matrix
    for user_id in user_item_matrix.index:
        # Get personalized recommendations for the current user using collab
        recommendations = get_recommendations(user_id, similarity_matrix, us

        # Print header for user recommendations
        print(f"\nTop recommendations for user {user_id}:")

        # Iterate over each recommended item and its score, and print the de
        for item_id, score in recommendations:
```

```
        for item_id, score in recommendations:
            print(f"Item {item_id} - Score: {score}")

# Main routine:
# The next function is the entry point for a recommendation system pipeline.
# It loads data from a CSV file, calculates Recency-Frequency-Monetary score
# for each customer, creates a user-item matrix representing interactions,
# computes cosine similarity between users, and finally prints personalized
# recommendations for each user.
#
# Parameters:
# - file_path: The file path to the CSV file containing the purchase data.

def main(file_path):
    # Load purchase data from the specified CSV file
    data = load_data(file_path)

    # Calculate RFM scores for each customer based on their purchase history
    rfm_data = calculate_rfm_scores(data)

    # Create a user-item matrix representing user-item interactions (purchas
    user_item_matrix = create_user_item_matrix(data)

    # Compute cosine similarity between users using the user-item matrix
    similarity_matrix = compute_cosine_similarity_from_scratch(user_item_mat

    # Print personalized recommendations for each user based on collaborativ
    print_recommendations(user_item_matrix, similarity_matrix, rfm_data)

if __name__ == "__main__":
    file_path = "./generated_data.csv"
    main(file_path)
```

```
Top recommendations for user 1:

Top recommendations for user 2:
Item 103 - Score: 20.80473229324013
Item 105 - Score: 14.905573806218525
Item 101 - Score: 9.50601847382327

Top recommendations for user 3:
Item 105 - Score: 22.486130523021266

Top recommendations for user 4:
Item 101 - Score: 16.985158618387658

Top recommendations for user 5:

Top recommendations for user 6:
Item 104 - Score: 22.074402147746678

Top recommendations for user 7:
Item 102 - Score: 23.95227400085748

Top recommendations for user 8:

Top recommendations for user 9:
Item 103 - Score: 22.234021672612403
Item 102 - Score: 12.886056705925558
Item 104 - Score: 12.85602987611598
```

```
Top recommendations for user 10:
```

In [2]:
```
# Bayesian Recommender System
#
# Igor Mol <igor.mol@makes.ai>
#
# The Bayesian Personalized Ranking (BPR) recommendation system, integrated
# with Recency, Frequency, Monetary (RFM) analysis, employs a collaborative
# filtering approach to provide personalized item recommendations. In this
# system, each user and item are represented by latent factors, which are
# learned during the training process.
#
# BPR focuses on optimizing the ranking of positive interactions (purchases)
# over negative interactions (non-purchases) for each user. The RFM analysis
# enhances the recommendation model by incorporating user-specific metrics
# related to the recency, frequency, and monetary value of their interactior
#
# By considering these RFM scores during training, the system adapts its
# recommendations to user preferences based not only on historical
# interactions but also on the user's specific purchase behavior patterns.
#
# We hope this hybrid approach may improve the accuracy and relevance of ite
# recommendations for individual users in the recommendation system.


import pandas as pd
import numpy as np

# The next function calculates RFM (Recency, Frequency, Monetary) scores for
# each customer based on their purchase history.
#
# Parameters:
# - data: DataFrame containing purchase history with columns 'customer_id',
#   'purchase_date', 'item_id', and 'amount'.
#
# Returns:
# - DataFrame with customer ID and unified RFM scores.

def calculate_rfm_scores(data):
    # Get the current date from the data
    current_date = data['purchase_date'].max()

    # Group data by customer and calculate recency, frequency, and monetary
    rfm_data = data.groupby('customer_id').agg({
        'purchase_date': lambda x: (current_date - x.max()).days,
        'item_id': 'count',
        'amount': 'sum'
    }).reset_index()

    # Normalize and score each metric
    rfm_data['recency_score'] = rfm_data['purchase_date'] / rfm_data['purcha
    rfm_data['frequency_score'] = rfm_data['item_id'] / rfm_data['item_id'].
    rfm_data['monetary_score'] = rfm_data['amount'] / rfm_data['amount'].max

    # Calculate unified RFM score by summing individual scores
    rfm_data['unified_rfm_score'] = rfm_data['recency_score'] + rfm_data['fr

    # Return DataFrame with customer ID and unified RFM scores
    return rfm_data[['customer_id', 'unified_rfm_score']]
```

```
# The function create_user_item_matrix() creates a user-item matrix from pur
# se history data, where rows represent users, columns represent items, and
# values indicate the number of times each user has interacted with each ite
#
# Parameters:
# - data: DataFrame containing purchase history with columns 'customer_id',
#   'item_id', and 'amount'.
#
# Returns:
# - User-item matrix with rows representing users, columns representing item
#   and values representing interaction counts.

def create_user_item_matrix(data):
    return pd.pivot_table(data, values='amount', index='customer_id', columr

# The sigmoid function returns σ(x) := 1/(1+e^(-x)) for a given input 'x'.

def sigmoid(x):
    return 1 / (1 + np.exp(-x))


# The function train_bpr trains a Bayesian Personalized Ranking (BPR) model
# stochastic gradient descent. It takes as input a user-item matrix, RFM dat
# and several optional hyperparameters. The goal is to learn latent factors
# users and items, incorporating RFM scores in the training process. The moc
# trained for a specified number of epochs, adjusting the latent factors to
# improve ranking performance.
#
# Parameters:
# - user_item_matrix: Matrix representing user-item interactions, where rows
#   are users and columns are items.
# - rfm_data: DataFrame containing RFM (Recency, Frequency, Monetary) data f
#   users.
# - learning_rate: Step size for gradient descent (default is 0.01).
# - regularization: Regularization term to prevent overfitting (default is 0
# - latent_factors: Number of latent factors for users and items (default is
# - epochs: Number of training epochs (default is 10).
# - rfm_weight: Weight of RFM score in the gradient calculation (default is

def train_bpr(user_item_matrix, rfm_data, learning_rate=0.01, regularizatior
    # Get the number of users and items in the user-item matrix
    num_users, num_items = user_item_matrix.shape

    # Initialize random latent factors for users and items
    user_latent_factors = np.random.rand(num_users, latent_factors)
    item_latent_factors = np.random.rand(num_items, latent_factors)

    # Iterate over training epochs
    for epoch in range(epochs):
        # Iterate over users in the user-item matrix
        for user_idx, row in user_item_matrix.iterrows():
            # Find items with positive interactions (purchased items)
            positive_items = np.where(row > 0)[0]
            if len(positive_items) == 0:
                continue

            # Randomly sample a positive item
            sampled_item = np.random.choice(positive_items)
```

```
            # Find items with no interactions (negative items)
            negative_items = np.where(row == 0)[0]
            if len(negative_items) == 0:
                continue

            # Randomly sample a negative item
            sampled_negative_item = np.random.choice(negative_items)

            # Calculate the difference in scores for the positive and negati
            x_uij = np.dot(user_latent_factors[user_idx], item_latent_factor
                np.dot(user_latent_factors[user_idx], item_latent_factors[sa

            sigmoid_x_uij = sigmoid(x_uij)

            # Include RFM score in the gradient calculation
            rfm_score = rfm_data[rfm_data['customer_id'] == user_idx]['unifi

            # Compute gradients for user and items
            user_gradient = (sigmoid_x_uij * (item_latent_factors[sampled_ne
                             item_latent_factors[sampled_item]) + regulariza
            item_gradient_positive = (sigmoid_x_uij * user_latent_factors[us
                                      regularization * item_latient_factors[sa
            item_gradient_negative = (-sigmoid_x_uij * user_latent_factors[u
                                      regularization * item_latient_factors[sa

            # Update latent factors using gradient descent
            user_latent_factors[user_idx] += learning_rate * user_gradient
            item_latent_factors[sampled_item] += learning_rate * item_gradie
            item_latent_factors[sampled_negative_item] += learning_rate * it

    # Return the learned user and item latent factors
    return user_latent_factors, item_latent_factors


# The next function predicts interaction scores for all users and items base
# the learned user/items relations.
#
# Parameters:
# - user_latent_factors: Latent factors for users learned during training.
# - item_latent_factors: Latent factors for items learned during training.
#
# Returns:
# - A matrix of predicted interaction scores for all users and items.

def predict_all(user_latent_factors, item_latent_factors):
    # Calculate the dot product of user and item latent factors
    return np.dot(user_latent_factors, item_latent_factors.T)


# The function recommend_top_k() recommends top k items for a given user bas
# on learned users/items relations. It considers the user's past interactior
# excludes items already purchased, and incorporates the user's RFM score to
# prioritize recommendations.
#
# Parameters:
# - user_id: The ID of the target user.
# - user_latent_factors: Latent factors for users learned during training.
# - item_latent_factors: Latent factors for items learned during training.
# - user_item_matrix: Matrix representing user-item interactions
```

```
# - user_item_matrix: Matrix representing user-item interactions.
# - rfm_data: DataFrame containing RFM (Recency, Frequency, Monetary) data.
# - k: Number of items to recommend (default is 5).
#
# Returns:
# - A list of top k recommended item indices.

def recommend_top_k(user_id, user_latent_factors, item_latent_factors, user_
    # Check if user_id exists in the user_item_matrix
    if user_id not in user_item_matrix.index:
        print(f"User {user_id} not found in the user_item_matrix.")
        return []

    # Get the positional index of the user in the user_item_matrix
    user_idx = user_item_matrix.index.get_loc(user_id)

    # Predict scores for all items for the target user
    user_scores = predict_all(user_latent_factors, item_latent_factors)[user

    # Identify items already purchased by the user
    already_purchased = np.where(user_item_matrix.loc[user_id].values > 0)[0
    user_scores[already_purchased] = -np.inf  # Set already purchased items

    # Retrieve unified RFM score for the user and multiply by the scores
    rfm_score_multiplier = rfm_data[rfm_data['customer_id'] == user_id]['uni
    user_scores *= rfm_score_multiplier  # Multiply by unified RFM score

    # Get the indices of the top k items based on scores
    top_k_items = np.argpartition(user_scores, -k)[-k:]

    # Return the top k items in descending order of scores
    return top_k_items[np.argsort(user_scores[top_k_items])][::-1]

# Loads data from a CSV file into a pandas DataFrame.

def load_data(file_path):
    return pd.read_csv(file_path)

# Preprocesses data by converting 'purchase_date' to datetime format and
# sorts the DataFrame by 'purchase_date' in descending order.

def preprocess_data(data):
    data['purchase_date'] = pd.to_datetime(data['purchase_date'])
    return data.sort_values(by='purchase_date', ascending=False)

# The main function calls the following routines:
#
# 1. Loads data from a CSV file located at "./generated_data.csv".
# 2. Preprocesses the loaded data.
# 3. Calculates RFM (Recency, Frequency, Monetary) scores for the preprocess
# 4. Creates a user-item matrix based on the preprocessed data.
# 5. Trains Bayesian Personalized Ranking (BPR) to obtain latent factors for
# 6. For each user in the user-item matrix, recommends the top items using t

def main():
    # Set the path to the CSV file containing the data
    file_path = "./generated_data.csv"

    # Step 1: Load data from the specified CSV file
    data = load_data(file_path)
```

```
    data = load_data(file_path)

    # Step 2: Preprocess the loaded data
    preprocessed_data = preprocess_data(data)

    # Step 3: Calculate RFM scores for the preprocessed data
    rfm_data = calculate_rfm_scores(preprocessed_data)

    # Step 4: Create a user-item matrix based on the preprocessed data
    user_item_matrix = create_user_item_matrix(preprocessed_data)

    # Step 5: Train BPR to obtain latent factors for users and items
    user_latent_factors, item_latent_factors = train_bpr(user_item_matrix, r

    # Step 6: For each user in the user-item matrix, recommend top items usi
    for user_id, _ in user_item_matrix.iterrows():
        top_recommendations = recommend_top_k(user_id, user_latent_factors,
        print(f"\nTop recommendations for user {user_id}: {top_recommendatic


if __name__ == "__main__":
    main()
```

```
Top recommendations for user 1: [4 3 2 1 0]

Top recommendations for user 2: [4 2 0 3 1]

Top recommendations for user 3: [4 3 2 1 0]

Top recommendations for user 4: [0 4 3 2 1]

Top recommendations for user 5: [4 3 2 1 0]

Top recommendations for user 6: [3 4 2 1 0]

Top recommendations for user 7: [1 4 3 2 0]

Top recommendations for user 8: [4 3 2 1 0]

Top recommendations for user 9: [1 2 3 4 0]

Top recommendations for user 10: [4 3 2 1 0]
```

In [3]:
```
# AutoREC: Personalized Recommender System with Autoencoder
#
# Igor Mol <igor.mol@makes.ai>
#
# In a recommender system, autoencoders are employed as a technique to model
# capture latent patterns in user-item interactions. An autoencoder consists
# an encoder and decoder that aim to compress and reconstruct input data,
# respectively. In the context of a recommender system, these autoencoders c
# learn meaningful representations of user preferences and item characterist
# RFM (Recency, Frequency, Monetary) analysis is a method commonly used in
# marketing to segment and understand customer behavior. When integrated wit
# autoencoder, RFM scores can serve as additional features during the traini
# process, enriching the model with information about the recency, frequency
# monetary aspects of user transactions. By combining RFM analysis with an
# autoencoder, the recommender system becomes capable of capturing both the
# inherent patterns in user-item interactions and the nuanced characteristic
# individual user behavior. This integration allows for more personalized ar
# effective recommendations by considering not only historical interactions
# also the specific context of each user's engagement.


import numpy as np
import pandas as pd

# Load and preprocess customer transaction data for RFM analysis

# Load the CSV file containing customer transaction data
file_path = "./generated_data.csv"
df = pd.read_csv(file_path)

# Convert 'purchase_date' column to datetime format for date manipulation
df['purchase_date'] = pd.to_datetime(df['purchase_date'])

# Encode customer and item IDs to sequential integers for efficient computat
df['customer_id'] = df['customer_id'].astype("category").cat.codes
df['item_id'] = df['item_id'].astype("category").cat.codes

# Create a user-item matrix representing customer transactions
# Rows represent customers, columns represent items, and values represent tr
user_item_matrix = pd.pivot_table(df, values='amount', index='customer_id',

# The data is now prepared for further analysis, including the calculation c

def calculate_rfm_scores(data):
    """
    Calculates RFM (Recency, Frequency, Monetary) scores for customer segmer

    Parameters:
    - data: Pandas DataFrame containing customer transaction data with colum
        - 'customer_id': Unique identifier for each customer.
        - 'purchase_date': Date of the purchase.
        - 'item_id': Identifier for each purchased item.
        - 'amount': Monetary value of each transaction.

    Returns:
    A Pandas DataFrame with columns 'customer_id' and 'unified_rfm_score':
    - 'customer_id': Unique identifier for each customer.
    - 'unified_rfm_score': Combined RFM score for each customer based on rec
    """
```

```
    # Find the most recent purchase date in the dataset
    current_date = data['purchase_date'].max()

    # Group the data by customer and calculate RFM metrics
    rfm_data = data.groupby('customer_id').agg({
        'purchase_date': lambda x: (current_date - x.max()).days,  # Calcula
        'item_id': 'count',  # Calculate frequency
        'amount': 'sum'  # Calculate monetary value
    }).reset_index()

    # Normalize the individual RFM scores to values between 0 and 1
    rfm_data['recency_score'] = rfm_data['purchase_date'] / rfm_data['purcha
    rfm_data['frequency_score'] = rfm_data['item_id'] / rfm_data['item_id'].
    rfm_data['monetary_score'] = rfm_data['amount'] / rfm_data['amount'].max

    # Combine the normalized RFM scores into a unified score for each custom
    rfm_data['unified_rfm_score'] = rfm_data['recency_score'] + rfm_data['fr

    # Return a DataFrame containing customer_id and the unified RFM score
    return rfm_data[['customer_id', 'unified_rfm_score']]


# Merge RFM scores with user-item matrix
rfm_data = calculate_rfm_scores(df)
user_item_matrix_with_rfm = pd.merge(user_item_matrix, rfm_data, on='custome

# Custom train-test split function from scratch

def train_test_split_scratch(data, test_size=0.2, random_state=None):
    """
    Split the input data into training and testing sets.

    Parameters:
    - data: Pandas DataFrame, the input dataset to be split.
    - test_size: float, optional (default=0.2), the proportion of the datase
    - random_state: int, optional (default=None), seed for reproducibility c

    Returns:
    Two Pandas DataFrames representing the training and testing sets:
    - The training set includes (1 - test_size) proportion of the input data
    - The testing set includes test_size proportion of the input data.
    """
    # Set the random seed for reproducibility
    np.random.seed(random_state)

    # Shuffle the indices of the input data
    shuffled_indices = np.random.permutation(len(data))

    # Calculate the number of samples for the test set
    test_size = int(len(data) * test_size)

    # Extract indices for the test and training sets
    test_indices = shuffled_indices[:test_size]
    train_indices = shuffled_indices[test_size:]

    # Return the training and testing sets based on the calculated indices
    return data.iloc[train_indices], data.iloc[test_indices]
```

```
# Data preparation and utility functions

# Split the user-item matrix with RFM scores into training and testing sets
train_data, test_data = train_test_split_scratch(user_item_matrix_with_rfm,

# Convert training and testing data to NumPy arrays
train_data_np = train_data.drop(columns=['unified_rfm_score']).values
train_rfm_scores = train_data['unified_rfm_score'].values.reshape(-1, 1)
test_data_np = test_data.drop(columns=['unified_rfm_score']).values

# Define the sigmoid activation function
def sigmoid(x):
    """
    Sigmoid activation function.

    Parameters:
    - x: NumPy array, input values.

    Returns:
    NumPy array, values transformed by the sigmoid function.
    """
    return 1 / (1 + np.exp(-x))

# Define the derivative of the sigmoid function
def sigmoid_derivative(x):
    """
    Derivative of the sigmoid activation function.

    Parameters:
    - x: NumPy array, input values.

    Returns:
    NumPy array, derivative values.
    """
    return x * (1 - x)

# Define the mean squared error loss function
def mean_squared_error(y_true, y_pred):
    """
    Mean squared error loss function.

    Parameters:
    - y_true: NumPy array, true values.
    - y_pred: NumPy array, predicted values.

    Returns:
    Float, mean squared error between true and predicted values.
    """
    return np.mean((y_true - y_pred)**2)


# Forward pass function for a simple autoencoder

def forward(x, weights_encoder, weights_decoder, bias_encoder, bias_decoder)
    """
    Perform the forward pass through a simple autoencoder.

    Parameters:
    - x: NumPy array, the input data for the autoencoder
```

```
    - x: NumPy array, the input data for the autoencoder.
    - weights_encoder: NumPy array, weights for the encoder layer.
    - weights_decoder: NumPy array, weights for the decoder layer.
    - bias_encoder: NumPy array, bias for the encoder layer.
    - bias_decoder: NumPy array, bias for the decoder layer.

    Returns:
    Two NumPy arrays:
    - x_hat: Output of the decoder, representing the reconstructed input.
    - z: Output of the encoder, representing the compressed latent space rep
    """
    # Apply the sigmoid activation function to the weighted sum of input and
    z = sigmoid(np.dot(x, weights_encoder) + bias_encoder)

    # Apply the sigmoid activation function to the weighted sum of encoder o
    x_hat = sigmoid(np.dot(z, weights_decoder) + bias_decoder)

    # Return the reconstructed input (x_hat) and the compressed latent space
    return x_hat, z

# Backward pass function for updating weights and biases in a simple autoenc

def backward(x, x_hat, z, weights_encoder, weights_decoder, bias_encoder, bi
    """
    Perform the backward pass to update weights and biases in a simple autoe

    Parameters:
    - x: NumPy array, the input data used during the forward pass.
    - x_hat: NumPy array, the reconstructed output from the forward pass.
    - z: NumPy array, the compressed latent space representation from the fo
    - weights_encoder: NumPy array, weights for the encoder layer.
    - weights_decoder: NumPy array, weights for the decoder layer.
    - bias_encoder: NumPy array, bias for the encoder layer.
    - bias_decoder: NumPy array, bias for the decoder layer.
    - learning_rate: float, the rate at which the weights and biases are upd
    - rfm_scores: NumPy array, additional RFM scores to be incorporated in t

    Returns:
    Updated weights and biases for both encoder and decoder layers:
    - weights_encoder: Updated weights for the encoder layer.
    - weights_decoder: Updated weights for the decoder layer.
    - bias_encoder: Updated bias for the encoder layer.
    - bias_decoder: Updated bias for the decoder layer.
    """
    # Calculate the error between the input and reconstructed output
    error = x - x_hat

    # Compute the delta for the decoder layer using the error and sigmoid de
    delta_decoder = error * sigmoid_derivative(x_hat)

    # Compute the delta for the encoder layer using the delta from the decod
    delta_encoder = delta_decoder.dot(weights_decoder.T) * sigmoid_derivativ

    # Update weights and biases for both decoder and encoder layers
    weights_decoder += learning_rate * z.T.dot(delta_decoder)
    bias_decoder += learning_rate * np.sum(delta_decoder, axis=0, keepdims=T
    weights_encoder += learning_rate * (x.T.dot(delta_encoder) + rfm_scores.
    bias_encoder += learning_rate * (np.sum(delta_encoder, axis=0, keepdims=

    # Return the updated weights and biases
```

```
    # Return the updated weights and biases
    return weights_encoder, weights_decoder, bias_encoder, bias_decoder


# Training loop for a simple autoencoder with RFM scores

def train_autoencoder(data, rfm_scores, input_dim, latent_dim, learning_rate
    """
    Train a simple autoencoder on the provided data with an optional inclusi

    Parameters:
    - data: Pandas DataFrame, the input dataset for training.
    - rfm_scores: Pandas DataFrame, RFM scores for each customer.
    - input_dim: int, the dimensionality of the input data.
    - latent_dim: int, the dimensionality of the compressed latent space rep
    - learning_rate: float, optional (default=0.001), the rate at which the
    - epochs: int, optional (default=20), the number of training epochs.
    - batch_size: int, optional (default=64), the size of each training batc

    Returns:
    Updated weights and biases for both encoder and decoder layers after tra
    - weights_encoder: Updated weights for the encoder layer.
    - weights_decoder: Updated weights for the decoder layer.
    - bias_encoder: Updated bias for the encoder layer.
    - bias_decoder: Updated bias for the decoder layer.
    """
    # Initialize weights and biases with random values
    weights_encoder = np.random.randn(input_dim, latent_dim)
    weights_decoder = np.random.randn(latent_dim, input_dim)
    bias_encoder = np.zeros((1, latent_dim))
    bias_decoder = np.zeros((1, input_dim))

    # Training loop
    for epoch in range(epochs):
        # Iterate through the data in batches
        for i in range(0, len(data), batch_size):
            # Extract batch data and corresponding RFM scores
            batch_data = data[i:i+batch_size].drop(columns=['unified_rfm_scc
            batch_rfm_scores = data[i:i+batch_size]['unified_rfm_score'].val

            # Perform forward pass and backward pass for weight and bias upc
            x_hat, z = forward(batch_data, weights_encoder, weights_decoder,
            weights_encoder, weights_decoder, bias_encoder, bias_decoder = \
                backward(batch_data, x_hat, z, weights_encoder, weights_decc

        # Print loss at the end of each epoch
        loss = mean_squared_error(data.drop(columns=['unified_rfm_score']).v
        print(f"Epoch {epoch+1}/{epochs}, Loss: {loss}")

    # Return the updated weights and biases after training
    return weights_encoder, weights_decoder, bias_encoder, bias_decoder


# Train the autoencoder with RFM scores
latent_dim = 64
learning_rate = 0.001
epochs = 20
batch_size = 64

weights_encoder, weights_decoder, bias_encoder, bias_decoder = \
```

```
    train_autoencoder(train_data, train_rfm_scores, input_dim=user_item_matr
                      learning_rate=learning_rate, epochs=epochs, batch_size

# Generate top recommendations for a specific user using an autoencoder mode

def get_top_recommendations(user_id, user_item_matrix, weights_encoder, weig
    """
    Generate top recommendations for a specific user using an autoencoder mo

    Parameters:
    - user_id: int, the identifier of the target user.
    - user_item_matrix: Pandas DataFrame, a user-item matrix representing hi
    - weights_encoder: NumPy array, weights for the encoder layer of the aut
    - weights_decoder: NumPy array, weights for the decoder layer of the aut
    - bias_encoder: NumPy array, bias for the encoder layer of the autoencod
    - bias_decoder: NumPy array, bias for the decoder layer of the autoencod

    Returns:
    A Pandas Series containing top recommendations for the specified user.
    Recommendations are sorted by predicted ratings in descending order.
    """
    # Extract the historical interaction data for the target user
    user_history = user_item_matrix.loc[user_id].drop('unified_rfm_score').v

    # Perform a forward pass to reconstruct the user's interaction data
    x_hat, _ = forward(user_history, weights_encoder, weights_decoder, bias_

    # Create a DataFrame with predicted ratings for items
    predicted_ratings = pd.DataFrame(x_hat, columns=user_item_matrix.columns

    # Sort items by predicted ratings in descending order to get top recomme
    recommendations = predicted_ratings.iloc[0].sort_values(ascending=False)

    # Return the top recommendations for the specified user
    return recommendations

# Print top recommendations for all users using the trained autoencoder mode

# Iterate through all users in the user-item matrix
for user_id in range(user_item_matrix.shape[0]):
    # Get top recommendations for the current user using the trained autoenc
    recommendations = get_top_recommendations(user_id, user_item_matrix_with
                                              weights_encoder, weights_decod

    # Select a specified number of top recommendations (adjust as needed)
    top_recommendations = recommendations.head(5)

    # Print the top recommendations for the current user
    print(f"\nTop Recommendations for User {user_id}:\n{top_recommendations}
```

```
Epoch 1/20, Loss: 2554.0249053311563
Epoch 2/20, Loss: 2548.015592292493
Epoch 3/20, Loss: 2542.8691580481827
Epoch 4/20, Loss: 2538.6779796024844
Epoch 5/20, Loss: 2535.779739347831
Epoch 6/20, Loss: 2532.9946743215974
Epoch 7/20, Loss: 2529.106410280779
Epoch 8/20, Loss: 2522.7379385761124
Epoch 9/20, Loss: 2516.073870878381
```

```
Epoch 10/20, Loss: 2514.463775448255
Epoch 11/20, Loss: 2512.90707071606
Epoch 12/20, Loss: 2511.4630273414355
Epoch 13/20, Loss: 2510.7739833284268
Epoch 14/20, Loss: 2510.4857743797706
Epoch 15/20, Loss: 2510.3293415019375
Epoch 16/20, Loss: 2510.228681117136
Epoch 17/20, Loss: 2510.157137367369
Epoch 18/20, Loss: 2510.1029734986664
Epoch 19/20, Loss: 2510.0601567760727
Epoch 20/20, Loss: 2510.025233756582

Top Recommendations for User 0:
2    0.999989
1    0.999938
0    0.999675
3    0.983667
4    0.981700
Name: 0, dtype: float64

Top Recommendations for User 1:
1    0.999977
4    0.999028
2    0.998674
3    0.298920
0    0.117329
Name: 0, dtype: float64

Top Recommendations for User 2:
2    0.999958
1    0.999594
4    0.999110
3    0.993176
0    0.981789
Name: 0, dtype: float64

Top Recommendations for User 3:
3    0.999974
2    0.999951
4    0.997178
0    0.992263
1    0.991119
Name: 0, dtype: float64

Top Recommendations for User 4:
3    0.999993
1    0.999904
2    0.999477
4    0.999166
0    0.997303
Name: 0, dtype: float64

Top Recommendations for User 5:
2    0.999911
0    0.999863
3    0.996444
4    0.993862
1    0.992399
Name: 0, dtype: float64
```

```
Top Recommendations for User 6:
0    0.999956
3    0.999951
1    0.999715
4    0.999607
2    0.977815
Name: 0, dtype: float64

Top Recommendations for User 7:
3    0.999971
2    0.999233
1    0.998981
0    0.998841
4    0.998575
Name: 0, dtype: float64

Top Recommendations for User 8:
0    0.999566
1    0.999502
2    0.999354
4    0.999175
3    0.985108
Name: 0, dtype: float64

Top Recommendations for User 9:
3    0.999993
1    0.998927
4    0.996225
0    0.992687
2    0.974064
Name: 0, dtype: float64
```

In [4]:
```
# Recommender System Using Graph Neural Network
#
# Igor Mol <igor.mol@makes.ai>
#
# In the following recommender system, a Graph Neural Network (GNN) is emplo
# together with the Recency, Frequency, and Monetary (RFM) analysis to model
# user-item interactions. The GNN architecture is designed to capture patter
# between users and items by incorporating user and item embeddings.
# These embeddings are learned through the network's training process, enabl
# model to understand and predict user behavior based on historical interact
# The RFM metrics, representing user engagement patterns, are integrated int
# GNN framework to enhance the accuracy and personalization of recommendatio
# combining the strengths of GNNs in capturing complex dependencies in
# graph-structured data and the insights provided by RFM metrics, this appro
# to provide more effective and tailored recommendations in diverse applicat
# domains such as e-commerce or content platforms.

import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split

# The function load_data loads data from a CSV file, processes it, and calcu
# Recency, Frequency, and Monetary (RFM) metrics for customer segmentation.
# Parameters:
#   - csv_file_path: Path to the CSV file containing the data
# Returns:
#   - data: Processed DataFrame with added RFM metrics
#   - user_mapping: Mapping of customer IDs to unique indices

def load_data(csv_file_path):
    # Read data from the CSV file
    data = pd.read_csv(csv_file_path)

    # Convert 'purchase_date' column to datetime format
    data['purchase_date'] = pd.to_datetime(data['purchase_date'])

    # Factorize 'customer_id' and 'item_id' columns to obtain indices and ma
    user_indices, user_mapping = pd.factorize(data['customer_id'])
    item_indices, item_mapping = pd.factorize(data['item_id'])

    # Add user and item indices to the DataFrame
    data['user_index'] = user_indices
    data['item_index'] = item_indices

    # Calculate Recency, Frequency, and Monetary metrics
    recency = (data['purchase_date'].max() - data['purchase_date']).dt.days
    frequency = data.groupby('customer_id')['purchase_date'].count()
    monetary = data.groupby('customer_id')['amount'].sum()

    # Map RFM metrics to the DataFrame based on 'customer_id'
    data['recency'] = data['customer_id'].map(recency)
    data['frequency'] = data['customer_id'].map(frequency)
    data['monetary'] = data['customer_id'].map(monetary)

    # Normalize the RFM metrics
    data['recency'] = data['recency'] / data['recency'].max()
```

```
    data['recency'] = data['recency'] / data['recency'].max()
    data['frequency'] = data['frequency'] / data['frequency'].max()
    data['monetary'] = data['monetary'] / data['monetary'].max()

    # Normalize and add 'timestamps' column based on purchase dates
    timestamps = data['purchase_date'].astype(np.int64) // 10**9
    timestamps_normalized = (timestamps - timestamps.min()) / (timestamps.ma
    data['timestamps'] = timestamps_normalized

    # Return processed data and user mapping
    return data, user_mapping

# The function create_gnn_model creates a Graph Neural Network (GNN) for pre
# Recency, Frequency, and Monetary (RFM) metrics based on user and item embe
# Parameters:
#   - num_users: Number of unique users in the dataset
#   - num_items: Number of unique items in the dataset
#   - embedding_size: Size of the user and item embeddings
#   - hidden_size: Size of the hidden layer in the model
# Returns:
#   - GNN model consisting of user embedding, item embedding, and two linear

def create_gnn_model(num_users, num_items, embedding_size, hidden_size):
    # Initialize user and item embeddings using nn.Embedding
    user_embedding = nn.Embedding(num_users, embedding_size)
    item_embedding = nn.Embedding(num_items, embedding_size)

    # Define the neural network layers: linear1, relu activation, linear2
    # Adjusted input size to include RFM metrics (2 * embedding_size + 4)
    linear1 = nn.Linear(2 * embedding_size + 4, hidden_size)
    relu = nn.ReLU()
    linear2 = nn.Linear(hidden_size, 3)  # Output 3 values for RFM metrics

    # Return the GNN model as a ModuleList
    return nn.ModuleList([user_embedding, item_embedding, linear1, relu, lir


# The function forward_pass performs a forward pass through the Graph Neural
# model to predict Recency, Frequency, and Monetary (RFM) metrics based on i
# Parameters:
#   - model: GNN model containing user and item embeddings, linear layers, a
#   - user_indices: Tensor of user indices for input embeddings
#   - item_indices: Tensor of item indices for input embeddings
#   - timestamps: Tensor of normalized timestamps for input features
#   - recency: Tensor of normalized recency values for input features
#   - frequency: Tensor of normalized frequency values for input features
#   - monetary: Tensor of normalized monetary values for input features
# Returns:
#   - Output tensor containing the predicted RFM metrics for the given input

def forward_pass(model, user_indices, item_indices, timestamps, recency, fre
    # Unpack model components
    user_embedding, item_embedding, linear1, relu, linear2 = model

    # Obtain user and item embeddings
    user_emb = user_embedding(user_indices)
    item_emb = item_embedding(item_indices)

    # Concatenate input features
    input_features = torch.cat([user_emb, item_emb, timestamps.unsqueeze(1)
```

```
    input_features = torch.cat([user_emb, item_emb, timestamps.unsqueeze(1),
                                recency.unsqueeze(1), frequency.unsqueeze(1)
                                monetary.unsqueeze(1)], dim=1)

    # Forward pass through linear layers and activation function
    h = linear1(input_features)
    h = relu(h)
    output = linear2(h)

    # Return the predicted RFM metrics
    return output


# The function train_model trains a Graph Neural Network (GNN) model using t
# training data, optimizing for Recency, Frequency, and Monetary (RFM) metri
# Parameters:
#   - model: GNN model to be trained
#   - data: DataFrame containing the input features and target RFM metrics
#   - criterion: Loss function for model optimization
#   - optimizer: Optimization algorithm for updating model parameters
#   - num_epochs: Number of training epochs (default is 20)
#   - test_size: Proportion of data to use for testing (default is 0.2)
#   - random_state: Seed for reproducibility (default is None)
# Returns:
#   - Trained GNN model
#   - Test data used for evaluation

def train_model(model, data, criterion, optimizer, num_epochs=20, test_size=
    # Split data into training and testing sets
    train_data, test_data = train_test_split_custom(data, test_size, random_

    # Training loop over epochs
    for epoch in range(num_epochs):
        # Perform forward pass to obtain predictions
        predictions = forward_pass(model, torch.LongTensor(train_data['user_
                                   torch.LongTensor(train_data['item_index'
                                   torch.FloatTensor(train_data['timestamps
                                   torch.FloatTensor(train_data['recency'].
                                   torch.FloatTensor(train_data['frequency'
                                   torch.FloatTensor(train_data['monetary']

        # Use RFM metrics as target variables
        targets = torch.FloatTensor(train_data[['recency', 'frequency', 'mor

        # Calculate loss and perform backpropagation
        loss = criterion(predictions, targets)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Print current epoch and loss
        print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {loss.item()}")

    # Return the trained model and the test data for evaluation
    return model, test_data


# The following function customizes the train-test split of a given DataFram
# allowing for a specified test size and random seed for reproducibility.
# Parameters:
```

```
# Parameters:
#   - data: DataFrame to be split into training and testing sets
#   - test_size: Proportion of data to use for testing (default is 0.2)
#   - random_state: Seed for reproducibility (default is None)
# Returns:
#   - train_data: Subset of the input data for training
#   - test_data: Subset of the input data for testing

def train_test_split_custom(data, test_size=0.2, random_state=None):
    # Set random seed if provided
    if random_state is not None:
        np.random.seed(random_state)

    # Create a mask to randomly assign data points to training or testing se
    mask = np.random.rand(len(data)) < 1 - test_size

    # Use the mask to create training and testing subsets
    train_data = data[mask]
    test_data = data[~mask]

    # Return the split datasets
    return train_data, test_data


# The following function generates predictions using the provided GNN model
# on the given test data, without performing any gradient calculations.
# Parameters:
#   - model: Trained GNN model for making predictions
#   - test_data: DataFrame containing the test data features
# Returns:
#   - test_predictions: Predictions for Recency, Frequency, and Monetary (RF
#                       on the provided test data

def make_predictions(model, test_data):
    # Disable gradient calculations during prediction
    with torch.no_grad():
        # Perform forward pass to obtain test predictions
        test_predictions = forward_pass(model, torch.LongTensor(test_data['u
                                        torch.LongTensor(test_data['item_inc
                                        torch.FloatTensor(test_data['timesta
                                        torch.FloatTensor(test_data['recency
                                        torch.FloatTensor(test_data['frequer
                                        torch.FloatTensor(test_data['monetar

    # Return the test predictions
    return test_predictions


# The next function generates and displays top recommendations for users
# based on predicted Recency, Frequency, and Monetary (RFM) metrics.
# Parameters:
#   - test_predictions: Predicted RFM metrics for test data
#   - user_mapping: Mapping of user indices to user IDs
#   - user_indices: Tensor of user indices for predictions
#   - data: DataFrame containing the original data
#   - num_users: Number of users for which recommendations are displayed (de
#   - use_rfm: Flag to indicate whether to use RFM metrics for recommendatio
# Returns: None (Prints top recommendations for users)

def get_top_recommendations(test_predictions, user_mapping, user_indices, da
```

```
def get_top_recommendations(test_predictions, user_mapping, user_indices, du
    # Use 'monetary' for ranking if RFM metrics are considered
    if use_rfm:
        top_recommendations = torch.argsort(test_predictions[:, 2], descendi
        print("\nTop Recommendations for All Users using RFM:")
    else:
        top_recommendations = torch.argsort(test_predictions.squeeze(), desc
        print("\nTop Recommendations for All Users without RFM:")

    # Display recommendations for each user
    for user_idx in top_recommendations:
        user_id = user_mapping[user_indices[user_idx.item()]]
        recommended_items = data.loc[data['user_index'] == user_indices[user
        print(f"User {user_id}: Recommended Items {recommended_items}")


def main():
    # Load data
    csv_file_path = "./generated_data.csv"
    data, user_mapping = load_data(csv_file_path)

    # Create and initialize the model
    num_users = len(user_mapping)
    num_items = len(data['item_id'].unique())
    embedding_size = 16
    hidden_size = 64
    model = create_gnn_model(num_users, num_items, embedding_size, hidden_si

    # Set up loss function and optimizer
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    # Train the model and make predictions
    model, test_data = train_model(model, data, criterion, optimizer)

    # Make predictions on the test set
    test_predictions = make_predictions(model, test_data)

    # Print top recommendations for all users
    get_top_recommendations(test_predictions, user_mapping, data['user_index

if __name__ == "__main__":
    main()
```

```
Epoch 1/20, Loss: 0.7755201458930969
Epoch 2/20, Loss: 0.7201178669929504
Epoch 3/20, Loss: 0.6672388315200806
Epoch 4/20, Loss: 0.6170300841331482
Epoch 5/20, Loss: 0.569197952747345
Epoch 6/20, Loss: 0.5236921310424805
Epoch 7/20, Loss: 0.48050999641418457
Epoch 8/20, Loss: 0.4396679401397705
Epoch 9/20, Loss: 0.40099936723709106
Epoch 10/20, Loss: 0.36451074481010437
Epoch 11/20, Loss: 0.33003315329551697
Epoch 12/20, Loss: 0.29765868186950684
Epoch 13/20, Loss: 0.26734888553619385
Epoch 14/20, Loss: 0.23911914229393005
Epoch 15/20, Loss: 0.21293488144874573
Epoch 16/20, Loss: 0.1887972354888916
```

```
Epoch 17/20, Loss: 0.16671542823314667
Epoch 18/20, Loss: 0.14663712680339813
Epoch 19/20, Loss: 0.1285434067249298
Epoch 20/20, Loss: 0.11250824481248856

Top Recommendations for All Users using RFM:
User 3: Recommended Items [102, 102, 104, 101, 101]
User 5: Recommended Items [104, 103, 101, 102, 102]
User 3: Recommended Items [102, 102, 104, 101, 101]
User 10: Recommended Items [103, 104, 103, 103, 105]
User 8: Recommended Items [105, 103, 103, 102, 105]
```

In [5]:
```
# k-NN Recommender System
#
# Igor Mol <igor.mol@makes.ai>
#
# The RFM (Recency, Frequency, Monetary) model in recommender systems analyz
# customer transaction data to calculate recency, frequency, and monetary
# scores. These scores contribute to a unified RFM score, providing a genera
# measure of a customer's engagement and spending behavior.
#
# In collaborative filtering, the k-NN (k-Nearest Neighbors) algorithm is us
# to produce recommendations. It computes the similarity between users or it
# based on their RFM scores. By identifying the k-nearest neighbors, the
# algorithm suggests items by considering the preferences of these neighbors
# This approach enhances personalized recommendations by utilizing the RFM
# model to capture user behavior patterns.

import pandas as pd
import numpy as np

# Define cosine similarity function
def cosine_measure(vector1, vector2):
    dot_product = sum(v1 * v2 for v1, v2 in zip(vector1, vector2))
    magnitude1 = sum(v ** 2 for v in vector1) ** 0.5
    magnitude2 = sum(v ** 2 for v in vector2) ** 0.5

    similarity = dot_product / (magnitude1 * magnitude2 + 1e-10)  # To avoid
    return similarity

# The calculate_unified_rfm_scores function computes unified RFM (Recency,
# Frequency, Monetary) scores for each customer in the provided DataFrame.
# The primary objectives include calculating recency, frequency, monetary va
# normalizing these values, and finally, combining them into a single unifie
# score.
#
# Parameters:
#   - df: The DataFrame containing customer transactions with columns like
#         'customer_id', 'purchase_date', 'item_id', and 'amount'.
#
# Returns:
#   - rfm_data: A DataFrame with customer-wise unified RFM scores, including
#               columns 'customer_id', 'recency', 'frequency', 'monetary', a
#               'unified_score'.
#
# Processing Steps:
#   - Determine the current date from the maximum purchase date in the DataF
#   - Group the data by customer_id and calculate recency, frequency, and mo
#     values.
#   - Normalize recency, frequency, and monetary values individually.
#   - Compute the unified RFM score as the average of the normalized recency
#     frequency, and monetary values.
#   - Return the resulting DataFrame with customer-wise unified RFM scores.

def calculate_unified_rfm_scores(df):
    current_date = df['purchase_date'].max()
    rfm_data = df.groupby('customer_id').agg({
        'purchase_date': lambda x: (current_date - x.max()).days,  # Recency
        'item_id': 'count',  # Frequency
        'amount': 'sum'  # Monetary
    }).reset_index()
```

```
    }).reset_index()

    # Normalize and create a unified RFM score
    rfm_data['recency'] = rfm_data['purchase_date'] / rfm_data['purchase_dat
    rfm_data['frequency'] = rfm_data['item_id'] / rfm_data['item_id'].max()
    rfm_data['monetary'] = rfm_data['amount'] / rfm_data['amount'].max()

    # Calculate unified RFM score
    rfm_data['unified_score'] = (rfm_data['recency'] + rfm_data['frequency']

    return rfm_data

# The calculate_similarity function computes the similarity matrix between
# users based on their unified RFM scores. The main objective is to measure
# cosine similarity between users, providing a pairwise similarity matrix.
#
# Parameters:
#   - rfm_data: A DataFrame containing customer-wise unified RFM scores with
#               columns like 'customer_id' and 'unified_score'.
#
# Returns:
#   - similarity_matrix: A 2D list representing the pairwise cosine similari
#                        between users. Each element (i, j) corresponds to t
#                        cosine similarity between users i and j.
#
# Computation Steps:
#   - Obtain the number of users in the DataFrame.
#   - Create a unified RFM matrix by reshaping the 'unified_score' column.
#   - Compute the pairwise cosine similarity between users using the cosine_
#     function, resulting in a similarity matrix.
#   - Return the computed similarity matrix.

def calculate_similarity(rfm_data):
    num_users = rfm_data.shape[0]
    unified_rfm_matrix = np.array(rfm_data['unified_score']).reshape(-1, 1)
    similarity_matrix = [[cosine_measure(unified_rfm_matrix[i], unified_rfm_
    return similarity_matrix

# The get_top_recommendations function generates top recommendations for eac
# user based on collaborative filtering using unified RFM scores. The goal i
# to predict user preferences for items the user has not interacted with.
#
# Parameters:
#   - user_item_matrix: A DataFrame representing user-item interactions, whe
#                       rows are users, columns are items, and values indica
#                       interactions (e.g., purchase history).
#   - rfm_similarity: A 2D list representing cosine similarity between users
#                     based on unified RFM scores.
#   - num_recommendations: The number of top recommendations to generate for
#                          user (default is 5).
#
# Returns:
#   - top_recommendations: A dictionary where keys are user IDs, and values
#                          lists of tuples containing item IDs and correspor
#                          predicted ratings. Each user's recommendations ar
#                          sorted in descending order of predicted ratings.
#
# Recommendation Generation Steps:
#   - Obtain the number of users and a list of all user IDs.
#   - Iterate over each user to predict ratings for items not yet interacte
```

```
#   - Iterate over each user to predict ratings for items not yet interacted
#   - Calculate predicted ratings using collaborative filtering based on
#     unified RFM scores.
#   - Sort the predictions in descending order of estimated ratings.
#   - Store the top recommendations for each user in a dictionary and returr

def get_top_recommendations(user_item_matrix, rfm_similarity, num_recommenda
    num_users = len(rfm_similarity)
    all_user_ids = user_item_matrix.index

    top_recommendations = {}
    for user_id in all_user_ids:
        # Get items the user has not bought yet
        items_to_predict = user_item_matrix.columns[~user_item_matrix.loc[us

        # Calculate predicted ratings for the items using unified RFM scores
        predicted_ratings = []
        for item_id in items_to_predict:
            numerator = sum(rfm_similarity[user_id][j] * user_item_matrix.il
            denominator = sum(rfm_similarity[user_id][j] for j in range(num_
            predicted_rating = numerator / (denominator + 1e-10)  # To avoid
            predicted_ratings.append((item_id, predicted_rating))

        # Sort predictions by estimated rating in descending order
        predicted_ratings.sort(key=lambda x: x[1], reverse=True)

        # Get top recommendations
        top_recommendations[user_id] = predicted_ratings[:num_recommendation

    return top_recommendations

# Main routine
def main():
    # Load the CSV file into a DataFrame
    csv_file_path = "./generated_data.csv"
    df = pd.read_csv(csv_file_path)

    # Convert 'purchase_date' to datetime format
    df['purchase_date'] = pd.to_datetime(df['purchase_date'])

    # Calculate unified RFM scores
    rfm_data = calculate_unified_rfm_scores(df)

    # Create a user-item matrix
    user_item_matrix = pd.pivot_table(df, values='amount', index='customer_i

    # Calculate cosine similarity between users based on unified RFM score
    rfm_similarity = calculate_similarity(rfm_data)

    # Get top recommendations for all users
    top_recommendations = get_top_recommendations(user_item_matrix, rfm_simi

    # Print top recommendations for all users
    for user_id, recommendations in top_recommendations.items():
        print(f"User {user_id}:")
        for item_id, rating in recommendations:
            print(f"  - Item {item_id}: Estimated Rating = {rating:.2f}")
        print()

if __name__ == "__main__":
```

```
if __name__ == "__main__":
    main()
```

```
User 1:

User 2:
  - Item 101: Estimated Rating = 51.67
  - Item 105: Estimated Rating = 46.40
  - Item 103: Estimated Rating = 36.04

User 3:
  - Item 105: Estimated Rating = 46.40

User 4:
  - Item 101: Estimated Rating = 51.67

User 5:

User 6:
  - Item 104: Estimated Rating = 44.76

User 7:
  - Item 102: Estimated Rating = 49.72

User 8:

User 9:
  - Item 102: Estimated Rating = 49.72
  - Item 104: Estimated Rating = 44.76
  - Item 103: Estimated Rating = 36.04

User 10:
```

In [6]:
```
# ALS-RFM Recommender Engine
#
# Igor Mol <igor.mol@makes.ai>

# calculate_item_factors Function:
# - Updates item factors using ALS.
# - Takes user factors, user-item interactions, RFM scores, regularization p
#   the number of latent factors as input.
# - Returns updated item factors.

# als_with_rfm Function:
# - Implements the ALS algorithm with unified RFM scores.
# - Takes a DataFrame containing user-item interactions and RFM scores, alor
#   parameters such as the number of latent factors, iterations, and regular
# - Returns user factors and item factors.

# generate_recommendations Function:
# - Generates recommendations using ALS factors.
# - Takes user factors, item factors, and the number of recommendations as i
# - Returns the top recommendations for each user.

# main Function:
# - Entry point of the program.
# - Loads data from a CSV file, performs preprocessing by calculating RFM sc
#   applies the ALS algorithm with unified RFM scores, generates recommendat
#   and prints the top recommendations for each user.


import pandas as pd
import numpy as np

def calculate_item_factors(relevant_user_factors, user_item_interactions, rf
    """
    Update item factors using ALS.

    Parameters:
    - relevant_user_factors: User factors corresponding to items the user ha
    - user_item_interactions: User-item interactions matrix
    - rfm_scores: Unified RFM scores for the items
    - lambda_reg: Regularization parameter
    - num_factors: Number of latent factors

    Returns:
    - Updated item factors
    """
    weighted_interactions = rfm_scores * user_item_interactions
    return np.linalg.solve(
        np.dot(relevant_user_factors.T, relevant_user_factors) + lambda_reg
        np.dot(relevant_user_factors.T, weighted_interactions)
    )

def als_with_rfm(df, num_factors=50, num_iterations=50, lambda_reg=0.1):
    """
    ALS algorithm with unified RFM scores.

    Parameters:
    - df: DataFrame containing user-item interactions and RFM scores
    - num_factors: Number of latent factors
    - num_iterations: Number of iterations for ALS training
```

```
    - num_iterations: Number of iterations for ALS training
    - lambda_reg: Regularization parameter

    Returns:
    - User factors and item factors
    """
    user_item_matrix_with_rfm = pd.pivot_table(df, values=['amount', 'rfm_ur
    user_item_array_with_rfm = user_item_matrix_with_rfm.values

    num_users, num_items = user_item_array_with_rfm.shape
    user_factors_with_rfm = np.random.rand(num_users, num_factors)
    item_factors_with_rfm = np.random.rand(num_items, num_factors)

    for _ in range(num_iterations):
        for i in range(num_users):
            user_rfm_scores = user_item_array_with_rfm[i, user_item_array_wi
            relevant_item_factors = item_factors_with_rfm[user_item_array_wi

            user_factors_with_rfm[i, :] = np.linalg.solve(
                np.dot(relevant_item_factors.T, relevant_item_factors) + lam
                np.dot(relevant_item_factors.T, user_rfm_scores)
            )

        for j in range(num_items):
            relevant_user_factors = user_factors_with_rfm[user_item_array_wi

            # Update: Corrected indexing for rfm_scores
            rfm_scores = user_item_array_with_rfm[user_item_array_with_rfm[:

            item_factors_with_rfm[j, :] = calculate_item_factors(
                relevant_user_factors,
                user_item_array_with_rfm[user_item_array_with_rfm[:, j] > 0,
                rfm_scores,
                lambda_reg,
                num_factors
            )

    return user_factors_with_rfm, item_factors_with_rfm

def generate_recommendations(user_factors, item_factors, num_recommendations
    """
    Generate recommendations using ALS factors.

    Parameters:
    - user_factors: User factors matrix
    - item_factors: Item factors matrix
    - num_recommendations: Number of recommendations to generate

    Returns:
    - Top recommendations for each user
    """
    num_users = user_factors.shape[0]
    top_recommendations = {}

    for i in range(num_users):
        user_recommendations = np.argsort(np.dot(user_factors[i, :], item_fa
        top_recommendations[i + 1] = user_recommendations + 1  # Adjust indi

    return top_recommendations
```

```
def main():
    # Load data and perform ALS with RFM
    csv_file_path = "./generated_data.csv"
    df = pd.read_csv(csv_file_path)
    df['purchase_date'] = pd.to_datetime(df['purchase_date'])

    max_purchase_date = df['purchase_date'].max()
    df['recency'] = (max_purchase_date - df['purchase_date']).dt.days
    rfm_df = df.groupby('customer_id').agg({
        'recency': 'min',
        'purchase_date': 'count',
        'amount': 'sum'
    }).reset_index()
    rfm_df.columns = ['customer_id', 'recency', 'frequency', 'monetary']
    rfm_df[['recency', 'frequency', 'monetary']] = (
        rfm_df[['recency', 'frequency', 'monetary']] - rfm_df[['recency', 'f
    ) / (rfm_df[['recency', 'frequency', 'monetary']].max() - rfm_df[['recer
    rfm_df['rfm_unified'] = rfm_df[['recency', 'frequency', 'monetary']].app
    df = pd.merge(df, rfm_df[['customer_id', 'rfm_unified']], on='customer_i

    user_factors_with_rfm, item_factors_with_rfm = als_with_rfm(df)

    # Generate and print recommendations
    top_recommendations_with_rfm = generate_recommendations(user_factors_wit
    for user_id, recommendations in top_recommendations_with_rfm.items():
        print(f"Top recommendations for user {user_id}: {recommendations}")

if __name__ == "__main__":
    main()
```

```
Top recommendations for user 1: [1 2 4 5 3]
Top recommendations for user 2: [2 1 5 4 3]
Top recommendations for user 3: [2 1 5 4 3]
Top recommendations for user 4: [2 5 1 4 3]
Top recommendations for user 5: [1 2 4 5 3]
Top recommendations for user 6: [5 2 1 4 3]
Top recommendations for user 7: [1 5 2 3 4]
Top recommendations for user 8: [1 2 4 5 3]
Top recommendations for user 9: [2 1 4 5 3]
Top recommendations for user 10: [2 5 1 3 4]
```

In [7]:
```
# PageRank Recommender System
#
# Igor Mol <igor.mol@makes.ai>
#
# This code implements a recommender system using customer transaction data.
# Key tasks include calculating RFM (Recency, Frequency, Monetary) scores,
# creating a bipartite graph connecting customers and items with RFM scores,
# computing PageRank scores for node importance, and printing top recommenda
#
# calculate_rfm_scores Function:
# - Calculates RFM scores for customers in the dataset.
# - Defines functions for recency, frequency, and monetary components.
# - Computes individual RFM components, normalizes them, and combines into a
#
# create_bipartite_graph Function:
# - Generates a bipartite graph from the dataset.
# - Constructs a dictionary representing the graph, connecting customers and
#   based on RFM scores. Nodes have types ('customer' or 'item') and neighbo
#
# compute_pagerank Function:
# - Calculates PageRank scores for nodes in a given graph.
# - Initializes and iterates PageRank scores, updating based on neighbors ur
#   convergence or a specified maximum number of iterations.
#
# print_top_recommendations Function:
# - Prints top recommendations for all users based on PageRank scores.
# - Takes a list of sorted customers and the bipartite graph as input.
#
# main Function:
# - Entry point of the program.
# - Loads dataset, calculates RFM scores, creates a bipartite graph,
#   computes PageRank scores, sorts customers, and prints top recommendation


import pandas as pd
import numpy as np

# The calculate_rfm_scores function computes RFM (Recency, Frequency, Moneta
# scores for each customer in the provided dataset and normalizes them.
# The main objectives are to calculate individual RFM components, normalize
# scores, and combine them into a single measure.
# Parameters:
#   - data: The input dataset containing customer transactions and informati
# Returns:
#   - None (The function modifies the input DataFrame in place).

def calculate_rfm_scores(data):
    # Define functions to calculate recency, frequency, and monetary compone
    def recency(date):
        today = pd.to_datetime('today')
        return (today - pd.to_datetime(date)).days

    def frequency(customer_id):
        return data[data['customer_id'] == customer_id].shape[0]

    def monetary(customer_id):
        return data[data['customer_id'] == customer_id]['amount'].sum()

    # Calculate individual RFM components for each customer
```

```
    # Calculate individual RFM components for each customer
    data['recency'] = data['purchase_date'].apply(recency)
    data['frequency'] = data['customer_id'].apply(frequency)
    data['monetary'] = data['customer_id'].apply(monetary)

    # Normalize the RFM scores
    data['recency_normalized'] = (data['recency'] - data['recency'].min()) /
    data['frequency_normalized'] = (data['frequency'] - data['frequency'].mi
    data['monetary_normalized'] = (data['monetary'] - data['monetary'].min()

    # Combine normalized RFM scores into a single measure using specified we
    data['rfm_score'] = (
        0.4 * data['recency_normalized'] +
        0.4 * data['frequency_normalized'] +
        0.2 * data['monetary_normalized']
    )

# The create_bipartite_graph function generates a bipartite graph from the
# provided dataset, where customers and items form two disjoint sets.
# The main objectives are to construct a bipartite graph with customer and
# item nodes, connecting them based on RFM scores.
# Parameters:
#   - data: The input dataset containing customer-item relationships and RFM
#     scores.
# Returns:
#   - G: A dictionary representing the bipartite graph. Each node has a 'typ
#     (either 'customer' or 'item') and 'neighbors' containing connected noc
#     with corresponding weights.
def create_bipartite_graph(data):
    # Create an empty dictionary to represent the bipartite graph
    G = {}

    # Iterate over rows in the input dataset to construct the graph
    for _, row in data.iterrows():
        customer_id = row['customer_id']
        item_id = row['item_id']
        weight = row['rfm_score']

        # If the customer node does not exist, add it to the graph
        if customer_id not in G:
            G[customer_id] = {'type': 'customer', 'neighbors': {}}

        # If the item node does not exist, add it to the graph
        if item_id not in G:
            G[item_id] = {'type': 'item', 'neighbors': {}}

        # Connect customer and item nodes with corresponding RFM score
        G[customer_id]['neighbors'][item_id] = weight
        G[item_id]['neighbors'][customer_id] = weight

    # Return the generated bipartite graph
    return G

# The compute_pagerank function calculates PageRank scores for nodes in a gi
# graph, representing the importance of each node based on its connections.
# The main objectives are to initialize and iterate PageRank scores until
# convergence.
# Parameters:
#   - G: The input graph represented as a dictionary with nodes, their types
#     and neighbors with corresponding weights
```

```
#     and neighbors with corresponding weights.
#   - alpha: Damping factor for PageRank calculation (default is 0.85).
#   - max_iter: Maximum number of iterations for PageRank convergence
#     (default is 100).
#   - tol: Tolerance level for convergence (default is 1e-6).
# Returns:
#   - pagerank_scores: A dictionary containing PageRank scores for each node
#     the graph.
def compute_pagerank(G, alpha=0.85, max_iter=100, tol=1e-6):
    # Initialize PageRank scores for each node
    pagerank_scores = {node: 1.0 for node in G}

    # Perform PageRank iterations
    for _ in range(max_iter):
        prev_pagerank_scores = pagerank_scores.copy()

        # Update PageRank scores for each node based on neighbors
        for node in G:
            sum_weighted_pageranks = sum(prev_pagerank_scores[neighbor] * G[
            pagerank_scores[node] = (1 - alpha) + alpha * sum_weighted_pager

        # Check for convergence using the tolerance level
        if np.sum(np.abs(np.array(list(pagerank_scores.values())) - np.array
            break

    # Return the final PageRank scores
    return pagerank_scores

def print_top_recommendations(sorted_customers, G):
    # Print top recommendations for all users
    for customer_id, score in sorted_customers:
        recommended_items = [neighbor for neighbor in G[customer_id]['neighb
        print(f"Customer {customer_id}: Top Recommendations - {recommended_i

def main():
    # Load the CSV file into a pandas DataFrame
    csv_file_path = "./generated_data.csv"
    data = pd.read_csv(csv_file_path)

    # Calculate RFM scores
    calculate_rfm_scores(data)

    # Create a bipartite graph
    G = create_bipartite_graph(data)

    # Compute PageRank scores
    pagerank_scores = compute_pagerank(G)

    # Sort customers based on PageRank scores
    sorted_customers = sorted(((k, v) for k, v in pagerank_scores.items() if

    # Print top recommendations
    print_top_recommendations(sorted_customers, G)

if __name__ == "__main__":
    main()
```

```
Customer 8: Top Recommendations - [101, 103, 102, 104, 105]
Customer 4: Top Recommendations - [102, 103, 104, 105]
Customer 1: Top Recommendations - [103, 104, 101, 105, 102]
```

```
Customer 10: Top Recommendations - [103, 101, 104, 105, 102]
Customer 3: Top Recommendations - [101, 102, 104, 103]
Customer 5: Top Recommendations - [103, 102, 105, 104, 101]
Customer 7: Top Recommendations - [103, 105, 104, 101]
Customer 6: Top Recommendations - [103, 105, 102, 101]
Customer 2: Top Recommendations - [104, 102]
Customer 9: Top Recommendations - [105, 101]
```

In [ ]: