Recommendation System - Project Writeup

Introduction

This writeup provides a detailed explanation of the Recommendation System code (Codenheimer(2).ipynb), developed by Team Codenheimer Laasya, Anirudh, Lahari, Tasneem) for the synthetic_user_dataset_10000.xlsx dataset. The system generates personalized recommendations for approximately 50 rewards, such as movie tickets, yoga sessions, and gardening kits, using TF-IDF vectorization, cosine similarity, and natural language processing NLP. The dataset contains 10,000 user records with columns: user_id, age_group, interests, activity_score, and past_rewards. This document breaks down each code component, detailing its purpose, implementation, and role in achieving a 99.4% accuracy rate on a 500-user evaluation subset.

Code Overview

The Recommendation System integrates content-based filtering (matching rewards to user interests) and collaborative filtering (leveraging similar users' past rewards) to deliver tailored recommendations. Key features include:

- Advanced NLP preprocessing to standardize and map interests to 20 broad c ategor ies.
- TF-IDF vectorization and sparse cosine similarity for efficient user matching.
- A user-reward matrix to track past rewards, with boosts for new rewards.
- Tiered recommendations based on activity score (basic: <50, standard: 50-74, premium: >=75.
- Diversity logic to ensure new user recommendations cover all input interests.
- Optimization via sparse matrices and caching for Google Colab compatibility.
-

The code is modular, scalable, and robust, addressing challenges like multi-word term standardization and invalid inputs. Below, each component is explained in depth with code snippets to illustrate its functionality.

Code Component Explanations

Library Imports

Snippet:

```
import pandas as pd
import numpy as np
import re
import nltk
import string
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.preprocessing import MinMaxScaler
import pickle
import os
import time
import random
from scipy import sparse
```

Library Imports

Snippet:

```
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
```

Explanation:

- pandas: Used for data manipulation and analysis, especially for handling data in DataFrame format.
- numpy: Provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.
- re: A module for working with regular expressions, useful for string manipulation and pattern matching.
- nltk: The Natural Language Toolkit, a library for working with human language data (text). It includes tools for text processing.
- string: Provides a set of string constants and utility functions.
- stopwords: A list of common words (like "and", "the", etc.) that are often removed from text data during preprocessing.
- WordNetLemmatizer: A tool for lemmatization, which reduces words to their base or root form.
- word_tokenize: A function to split text into individual words or tokens.
- TfidfVectorizer: Converts a collection of raw documents to a matrix of TF-IDF features, which helps in text analysis.
- cosine_similarity: A function to compute the cosine similarity between two non-zero vectors, often used in recommendation systems.

- MinMaxScaler: A preprocessing technique that scales features to a given range, typically [0, 1].
- pickle: A module for serializing and de-serializing Python objects, useful for saving models or data structures.
- os: Provides a way of using operating system-dependent functionality like reading or writing to the file system.
- time: A module to work with time-related functions.
- random: A module that implements pseudo-random number generators for various distributions.
- sparse: A module from scipy that provides functions for working with sparse matrices.

Load the Dataset

Snippet:

```
FILE_PATH = "synthetic_user_dataset_10000.xlsx"
df = pd.read_excel(FILE_PATH)
```

Explanation:

- This section loads the dataset from an Excel file into a pandas DataFrame.
- FILE_PATH: Specifies the location of the dataset.
- pd.read_excel: Reads the Excel file and creates a DataFrame, which is a 2-dimensional labeled data structure with columns of potentially different types.

Define Term Mappings

Snippet:

```
term_mappings = {
    'tool set': 'tool_set', 'tool kit': 'tool_kit', 'game credit': 'game_credit', 'game DLC': 'game_DLC',
    'concert pass': 'concert_pass', 'zoo ticket': 'zoo_ticket', 'free meal': 'free_meal', 'flight deal': 'flight_deal',
    'sports equipment': 'sports_equipment', 'cooking class': 'cooking_class', 'dance lesson': 'dance_lesson',
    'music festival': 'music_festival', 'movie premiere': 'movie_premiere', 'pet accessory': 'pet_accessory',
    'travel guide': 'travel_guide', 'news subscription': 'news_subscription', 'gaming console': 'gaming_console',
    'diy workshop': 'diy_workshop', 'vip pass': 'vip_pass', 'streaming voucher': 'streaming_voucher',
    'book club': 'book_club', 'food tour': 'food_tour', 'pet training': 'pet_training', 'craft kit': 'craft_kit',
    'chef workshop': 'chef_workshop', 'movie ticket': 'movie_ticket', 'concert ticket': 'concert_ticket',
    'sports ticket': 'sports_ticket', 'food coupon': 'food_coupon', 'news coupon': 'news_coupon',
    'travel voucher': 'travel_voucher', 'music workshop': 'music_workshop',
    'art class': 'art_class', 'photo workshop': 'photo_workshop', 'writing course': 'writing_course',
    'gardening kit': 'gardening_kit', 'tech gadget': 'tech_gadget', 'fashion voucher': 'fashion_voucher',
    'museum pass': 'museum_pass', 'yoga session': 'yoga_session', 'board game': 'board_game'
}
```

Explanation:

- This dictionary maps specific phrases to standardized terms.
- For example, "tool set" is mapped to "tool_set". This helps in normalizing the data by ensuring that different variations of the same term are treated consistently.

Function to Standardize Terms

Snippet:

```
def standardize_terms(text):
    if not isinstance(text, str):
        return text
    for term, replacement in term_mappings.items():
        text = text.replace(term, replacement)
    return text
```

Explanation:

- This function takes a string input and replaces any terms found in the term_mappings dictionary with their standardized versions.
- It first checks if the input is a string; if not, it returns the input unchanged.

Preprocess Interests and Past Rewards

Snippet:

```
df['interests'] = df['interests'].apply(lambda x: ' '.join(str(x).split(',')) if pd.notnull(x) else '')
df['past_rewards'] = df['past_rewards'].apply(lambda x: ' '.join(str(x).split(',')) if
pd.notnull(x) else '')
df['interests'] = df['interests'].apply(standardize_terms)
df['past_rewards'] = df['past_rewards'].apply(standardize_terms)
df['interests_original'] = df['interests']
df['past_rewards_original'] = df['past_rewards']
```

Explanation:

- This section preprocesses the interests and past_rewards columns in the DataFrame:
  - It replaces commas with spaces to create a single string of interests or rewards.
  - It applies the standardize_terms function to ensure all terms are standardized.
  - It creates original columns to keep the unmodified versions of interests and past rewards for reference.

Define Interest Mappings

Snippet:

```
interest_mappings = {
    'movies': 'movies', 'movie': 'movies', 'gaming': 'gaming', 'game': 'gaming', 'music': 'music',
    'books': 'books', 'book': 'books', 'reading': 'books',
    'diy': 'crafts', 'crafting': 'crafts', 'woodworking': 'crafts', 'knitting': 'crafts',
    'animals': 'pets', 'animal': 'pets', 'dog': 'pets', 'cat': 'pets', 'pet': 'pets',
    'news': 'current_events', 'current events': 'current_events', 'politics': 'current_events',
    'dance': 'dancing', 'dancing': 'dancing', 'ballet': 'dancing', 'salsa': 'dancing',
    'travel': 'travel', 'travelling': 'travel', 'vacation': 'travel',
    'food': 'cooking', 'cooking': 'cooking', 'baking': 'cooking', 'eating': 'cooking',
    'sports': 'sports', 'sport': 'sports', 'fitness': 'sports', 'exercise': 'sports',
    'jogging': 'sports', 'jog': 'sports', 'running': 'sports', 'hiking': 'sports',
    'swimming': 'sports', 'swim': 'sports', 'working out': 'sports', 'work out': 'sports',
    'workout': 'sports', 'weightlifting': 'sports', 'gym': 'sports', 'sprinting': 'sports', 'sprint':
'sports',
    'art': 'art', 'painting': 'art', 'drawing': 'art', 'sculpting': 'art',
    'photography': 'photography', 'photo': 'photography', 'camera': 'photography',
    'writing': 'writing', 'creative writing': 'writing', 'journaling': 'writing',
    'gardening': 'gardening', 'plants': 'gardening', 'landscaping': 'gardening',
    'tech': 'tech', 'technology': 'tech', 'coding': 'tech', 'gadgets': 'tech',
    'fashion': 'fashion', 'clothing': 'fashion', 'style': 'fashion', 'design': 'fashion',
    'history': 'history', 'historical': 'history', 'archaeology': 'history',
    'yoga': 'yoga', 'meditation': 'yoga', 'wellness': 'yoga',
    'board games': 'board_games', 'tabletop': 'board_games', 'strategy games':
'board_games'
}
```

Explanation:

- This dictionary maps various forms of interests to a standardized category.
- For example, both "movie" and "movies" are mapped to "movies". This helps in grouping similar interests together for analysis.

Function to Map Interests

Snippet:

```
def map_interests(text):
    tokens = text.lower().split()
    mapped = [interest_mappings.get(token, token) for token in tokens]
    return ' '.join(mapped)
```

Explanation:

- This function takes a string of interests, converts it to lowercase, splits it into tokens (words), and maps each token to its standardized form using the interest_mappings dictionary - It returns a single string of mapped interests, ensuring consistency in how interests are represented.

Map Interests in DataFrame

Snippet:

```
df['interests_mapped']
= df['interests_original'].apply(map_interests)
```

Explanation:

- This line applies the map_interests function to the interests_original column, creating a new column interests_mapped that contains the standardized interests for each user.

Fill Missing Values in Selected Features

Snippet:

```
features_selected = ['age_group', 'interests', 'activity_score', 'past_rewards']
for feature in features_selected:
    if feature == 'activity_score':
        df[feature] = df[feature].fillna(df[feature].mean())
    else:
        df[feature] = df[feature].fillna('unknown')
```

Explanation:

- This section fills in missing values for selected features:
  - For activity_score, it replaces missing values with the mean of the existing scores.
  - For other features, it fills missing values with the string 'unknown', ensuring that the DataFrame remains complete for analysis.

Preprocess Text Function

Snippet:

```
stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()

def preprocess_text(text, preserve_interests=False):
    if not isinstance(text, str) or not text.strip():
        return ''
    text = text.lower()
    text = re.sub(r'\b(?:and|or|&|\|)\b', ' ', text)
    text = re.sub(r'[,]', ' ', text)
    text = re.sub(r'\d+', ' ', text)
    text = re.sub(f"[{re.escape(string.punctuation)}]", " ", text)
    tokens = word_tokenize(text)
    tokens = [word for word in tokens if word not in stop_words]
    if preserve_interests:
        tokens = [interest_mappings.get(word, word) for word in tokens]
    return ' '.join([lemmatizer.lemmatize(word) for word in tokens])
```

Explanation:

- This function preprocesses text by:
  - Converting it to lowercase.
  - Removing conjunctions, punctuation, and numbers using regular expressions.
  - Tokenizing the text into individual words.
  - Filtering out stop words.
  - Optionally preserving interest mappings if specified.
  - Lemmatizing the remaining words to their base forms before returning the cleaned text.

Preprocess Columns in DataFrame

Snippet:

```
def preprocess_column(col):
    col = col.str.lower() \
            .replace(r'\b(?:and|or|&|\|)\b', ' ', regex=True) \
            .replace(r'[,]', ' ', regex=True) \
            .replace(r'\d+', ' ', regex=True) \
            .replace(f"[{re.escape(string.punctuation)}]", " ", regex=True)
    return col.apply(lambda x: ' '.join([lemmatizer.lemmatize(w) for w in
word_tokenize(x) if w not in stop_words]) if isinstance(x, str) else '')

df['age_group'] = preprocess_column(df['age_group'])
df['interests'] = preprocess_column(df['interests'])
df['past_rewards'] = preprocess_column(df['past_rewards'])
df['interests'] = df['interests'].apply(lambda x: '
'.join([interest_mappings.get(w, w) for w in x.split()]) if x else '')
```

Explanation:

- This section defines a function to preprocess specific DataFrame columns by:
  - Lowercasing the text.
  - Removing conjunctions, punctuation, and numbers.
  - Lemmatizing the words while filtering out stop words.
- It applies this function to the age_group, interests, and past_rewards columns, ensuring they are cleaned and standardized.

Normalize Activity Score

Snippet:

```
scaler = MinMaxScaler()
df['activity_score_norm'] = scaler.fit_transform(df[['activity_score']])
```

Explanation:

- This section normalizes the activity_score using Min-Max scaling, which transforms the scores to a range between 0 and 1.
- The normalized scores are stored in a new column activity_score_norm, making it easier to compare activity levels across users.

Load or Compute User Similarity Matrix

Snippet:

```
CACHE_PATH = "similarity_matrix_fixed.pkl"

vectorizer = TfidfVectorizer(ngram_range=(1, 4), max_df=0.85, max_features=5000)
feature_vectors = vectorizer.fit_transform(combined_features)

if os.path.exists(CACHE_PATH):
    with open(CACHE_PATH, 'rb') as f:
        user_similarity = pickle.load(f).toarray()
else:
    user_similarity = cosine_similarity(feature_vectors)
    sparse_similarity = sparse.csr_matrix(user_similarity)
    with open(CACHE_PATH, 'wb') as f:
        pickle.dump(sparse_similarity, f)
```

Explanation:

- This section checks if a cached similarity matrix exists. If it does, it loads the matrix from a file.
- If not, it computes the cosine similarity between the feature vectors generated from the combined features using TfidfVectorizer, which converts the text data into a matrix of TF-IDF features.
- The resulting similarity matrix is then saved to a file for future use, optimizing performance by avoiding repeated calculations.

Create User Reward Matrix

Snippet:

```
all_rewards = set()

for rewards in df['past_rewards_original']:
    if pd.notnull(rewards):
        reward_list = [r.strip() for r in re.split(r'[,\s]+', rewards) if r.strip()]
        cleaned_rewards = []
        i = 0
        while i < len(reward_list):
                        if i + 1 < len(reward_list) and f"{reward_list[i]}
{reward_list[i+1]}" in term_mappings.values():
                                                cleaned_rewards.append(f"
{reward_list[i]}_{reward_list[i+1]}")
                i += 2
            else:
                cleaned_rewards.append(reward_list[i])
                i += 1
        all_rewards.update(cleaned_rewards)

all_rewards.update(reward_interest_map.keys())
all_rewards = sorted([r for r in all_rewards if r in
reward_interest_map])

user_reward_matrix = np.zeros((len(df), len(all_rewards)))
reward_to_idx = {reward: idx for idx, reward in
enumerate(all_rewards)}

for idx, rewards in enumerate(df['past_rewards_original']):
    if pd.notnull(rewards):
        reward_list = [r.strip() for r in re.split(r'[,\s]+', rewards) if r.strip()]
        cleaned_rewards = []
        i = 0
        while i < len(reward_list):
                        if i + 1 < len(reward_list) and f"{reward_list[i]}
{reward_list[i+1]}" in term_mappings.values():
                                                cleaned_rewards.append(f"
{reward_list[i]}_{reward_list[i+1]}")
                i += 2
            else:
                cleaned_rewards.append(reward_list[i])
                i += 1
        for reward in cleaned_rewards:
            if reward in reward_to_idx:
                user_reward_matrix[idx, reward_to_idx[reward]] = 1
```

Explanation:

- This section creates a user reward matrix that indicates which rewards each user has received.
- It first compiles a set of all unique rewards from the past_rewards_original column, cleaning the data to ensure consistency.
- It then initializes a matrix where rows represent users and columns represent rewards, filling in the matrix based on the rewards each user has received.

Function to Recommend Rewards

Snippet:

```
def recommend_rewards(user_idx, num_recommendations=3):
    sim_scores = user_similarity[user_idx]
    threshold = 0.3
     filtered_users = [i for i, score in enumerate(sim_scores) if score > threshold and i != user_idx]

    if not filtered_users:
        filtered_users = np.argsort(sim_scores)[::-1][1:11]
    else:
                filtered_users = sorted(filtered_users, key=lambda x: sim_scores[x], reverse=True)[:10]

    reward_scores = np.zeros(len(all_rewards))

    for sim_user in filtered_users:
                    reward_scores += user_reward_matrix[sim_user] * sim_scores[sim_user]

    user_interests = df.iloc[user_idx]['interests_mapped'].split()

    for reward, idx in reward_to_idx.items():
        reward_interests = reward_interest_map.get(reward, [])
        for interest in reward_interests:
            if interest in user_interests:
                reward_scores[idx] *= 300.0

    if reward_scores.max() < 0.5:
        for reward, idx in reward_to_idx.items():
            reward_interests = reward_interest_map.get(reward, [])
            for interest in reward_interests:
                if interest in user_interests:
                    reward_scores[idx] += 300.0

                        top_indices = np.argsort(reward_scores)[::-1][:num_recommendations]
    recommended_rewards = [all_rewards[idx] for idx in top_indices]

    activity = df.iloc[user_idx]['activity_score']
     tier = 'premium' if activity >= 75 else 'standard' if activity >= 50 else 'basic'

    return recommended_rewards, tier
```

Explanation:

- This function generates reward recommendations for a specific user based on their interests and the similarity scores of other users.
- It first retrieves the similarity scores for the user and identifies other users with a similarity score above a defined threshold.
- It calculates reward scores based on the rewards received by similar users, adjusting scores for rewards that match the user's interests.
- If the maximum score is low, it boosts scores for rewards that align with the user's interests.
- Finally, it selects the top recommended rewards and determines the user's tier based on their activity score.

Function to Simulate Accuracy of Recommendations

Snippet:

```
def simulate_accuracy(user_idx, recommended_rewards):
    user_interests = set(df.iloc[user_idx]['interests_mapped'].split())
        user_past_rewards = set([r.strip() for r in re.split(r'[,\s]+', df.iloc[user_idx]['past_rewards_original']) if r.strip()])

    for reward in recommended_rewards:
        reward_interests = set(reward_interest_map.get(reward, []))
            if reward_interests & user_interests or reward in user_past_rewards:
                return True

    return False
```

Explanation:

- This function checks the accuracy of the recommendations by comparing the recommended rewards against the user's interests and past rewards.
- It returns True if any of the recommended rewards match the user's interests or if they have previously received those rewards, indicating a successful recommendation.

Evaluate Model Accuracy

Snippet:

```
Accuracy_scores = []
eval_subset = df.sample(n=500, random_state=42).index

for user_idx in eval_subset:
    recommendations, tier = recommend_rewards(user_idx)
    satisfied = simulate_accuracy(user_idx, recommendations)
    Accuracy_scores.append(satisfied)

accuracy_rate = np.mean(Accuracy_scores) * 100
print(f"Model Accuracy: {accuracy_rate:.2f}%")
```

Explanation:

- This section evaluates the accuracy of the recommendation model by sampling a subset of users.
- For each user, it generates recommendations and checks if they are satisfied using the simulate_accuracy function.
- It calculates the overall accuracy rate as the percentage of satisfied recommendations and prints the result.

Conclusion

- Developed by **Team Codenheimer**
- Members:
  - **Laasya**
  - **Anirudh**
  - **Lahari**
  - **Tasneem**
- the Recommendation System delivers highly accurate and diverse reward recommendations through a sophisticated pipeline of NLP preprocessing, TF-IDF vectorization, and hybrid filtering.
- Its modular design, optimized with sparse matrices and caching, ensures scalability and performance for 10,000 users.
- Model Accuracy = 99.4%