Marche Polytechnic University
Department of Information Engineering

Tech Recommender System: Customized Component Suggestions and Management

Project Members: Francalancia Simone

Professor: Endrit Xhina

Professor’s assistant: Anxhela Kosta

Master's Degree in Computer Science and Automation Engineering

September 7, 2023

Index

Project topic overview and goal

This software defines a very basic PC components recommender system and provides various functionalities for users and administrators. Its main goal is to allow users to **get component recommendations** based on their search history, price range, selected category and similar user interests.

Users can also review components and check their own reviews; administrators can view all facts, insert new components, and delete existing components from the knowledge base.

Here's a brief summary of the main functionalities:

1. **Main Menu:**
    - Allows users to switch between the user and admin menus or exit the program.

2. **User Menu:**
    - Get component recommendations for a user based on search history, price range, and selected category.
    - Review components and assign ratings.
    - Check user reviews for components.

3. **Admin Menu:**
    - View all facts in the knowledge base (categories, prices, user ratings).
    - Insert new components into the knowledge base with category and price information.
    - Delete existing components from the knowledge base.

The source file of the recommender system is available in my GitHub repository at this link: https://github.com/Simo-univpm/proj-ia

What is a recommender system and what are its applications

A recommender system, also known as a recommendation system or a recommendation engine, is a software or algorithmic system designed to provide personalized suggestions or recommendations to users. These recommendations are typically based on the user's preferences, historical behavior, or other relevant data. The primary goal of a recommender system is to **help users discover items or content they might be interested in**, thereby improving user experience and engagement.

Recommender systems find applications in various domains and industries due to their ability to enhance user satisfaction, increase sales, and optimize content delivery. Some common applications of recommender systems include:

- **E-commerce and Retail:**
    - Recommending products to online shoppers based on their browsing and purchase history
    - Suggesting related or complementary items to encourage upselling and cross-selling

- **Online Advertising:**
    - Delivering targeted ads to users based on their interests and online behavior
    - Improving click-through rates and ad relevance

- **Social Media:**
    - Recommending connections, friends, or groups to users on social networking platforms
    - Showing relevant posts, articles, or content in a user's feed

- **Gaming:**
    - Recommending video games or in-game items to players
    - Enhancing player experiences and in-game purchases

- **Education:**
    - Recommending online courses, learning resources, or textbooks to students
    - Personalizing educational content to match individual learning styles

Recommender systems leverage various techniques, including **collaborative filtering** (user-based or item-based), content-based filtering, matrix factorization, deep learning models, and hybrid approaches to generate recommendations. These systems are a key component of many online platforms and services, helping users discover relevant content and products while contributing to business growth and customer satisfaction.

Project description

The main concept on which the project relies on is a **user-based collaborative filtering approach**, which is designed to provide users with personalized recommendations based on the preferences and behaviors of **similar users**. The fundamental idea behind the User-Based Collaborative Filtering is that if two users have similar tastes or preferences in the past, they are likely to have similar tastes in the future.

Here is a brief explanation of the approach followed by this system:

- **Step 1: User-Item Matrix:** The system starts by building a user-item matrix, where **rows** represent **users**, **columns** represent **items** (in this case, electronic components), and each **cell** contains the user's **rating** for an item if available.

- **Step 2: Finding Similar Users:** To make recommendations for a particular user, the system identifies other users who have rated or interacted with similar items. This is done by calculating the similarity between users based on their past ratings. **Pearson correlation** is used in this system to measure similarity between users.

- **Step 3: Generating Recommendations:** Once similar users are identified, the system generates recommendations for the target user based on what the similar users have interacted with positively. This means that if User A and User B have similar preferences and User B has rated an item highly, User A may be recommended that item.

- **Step 4: Filtering and Ranking:** Recommendations are filtered and ranked before being presented to the user. Filtering can include removing items the user has already interacted with or items outside a specified price range. The remaining items are ranked based on predicted user preferences.

- **Step 5: Presenting Recommendations:** Finally, the system presents the recommended items to the user.

In summary, the collaborative filtering approach used in this recommendation system leverages the past behavior and preferences of similar users to generate personalized recommendations. It is a user-centric approach that doesn't rely on detailed item information.

To provide suggestions, the system will need facts and rules. The defined facts within the system are about users, products, product prices, user reviews, and user search histories of users. It is important to notice that every fact, except the “user” and “search_history”, is declared “**dynamic**” so that admins can operate and **edit** the knowledge base.

Facts:

- `User/1`: Facts that specify the **registered users** in the recommendation system.
- `Search_history/2`: Facts that store the **search history of registered users**, it is used to understand their past preferences and behavior. It helps personalize recommendations by considering what products a user has searched for in the past. It has two arguments: the username and a list of components they have searched for.
- `Category/2`: A dynamic predicate used to store the **category** (CPU, GPU, RAM) **of components.** It has two arguments: the component model and its category.
- `Price/2`: A dynamic predicate used to store the **price of components**. It has two arguments: the component model and its price.
- `User_rating/3`: A dynamic predicate used to store user ratings for electronic components. It has three arguments: the username, component model, and rating (float value).

Alongside with the facts, there are **three main rules** which **governates the system** and connects all the various components together, these rules are central to the functionality of the recommendation system. **init/0** manages user interaction, **pearson_correlation/3** calculates similarity between users, and **recommend/5** generates personalized product recommendations based on user input and behavior.

Rules:

- `init/0`: entry point for the recommendation system. It is responsible for **displaying** the main **menu** to the user and handling user input. The main menu provides options for the user to navigate to different sections of the recommendation system, such as the user menu or admin menu. Depending on the user's choice, `init/0` invokes the appropriate menu and functionality for the selected option. It also includes an option to exit the program.
- `pearson_correlation/3`: Pearson correlation is used to measure the **similarity between two users** based on their product ratings. It helps identify users with similar preferences, which is essential for collaborative filtering. It takes three arguments: `RatingsUserA`, `RatingsUserB`, and `Score`. `RatingsUserA` and `RatingsUserB` are lists of ratings given by two different users for a set of items. `Score` represents the calculated Pearson correlation coefficient, which measures the linear relationship between the ratings of the two users. The rule calculates the correlation based on the sum of products of ratings, the sum of squared ratings, and the length of the rating lists.

- `recommend/5`: this is the main predicate that keeps it all together. It **generates product recommendations** for a user based on their search history, preferences, and user ratings. It takes five arguments: `User`, `MinPrice`, `MaxPrice`, `SelectedCategory`, and `Recommendations`. `User` represents the username of the user for whom recommendations are generated. `MinPrice` and `MaxPrice` specify the minimum and maximum price range for the recommended products. `SelectedCategory` is the category of products the user is interested in (e.g., CPU, GPU, RAM), or it can be set to "skip" to let the system choose a category). `Recommendations` is the list of recommended products that will be generated by the rule. The rule proceeds by finding matching products that fit the user's criteria, considering user ratings, identifying similar users, generating collaborative recommendations, and finally formatting and presenting the recommendations to the user.

Flow of the program:

The main logic flow to get a recommendation involves gathering user input, identifying matching products, considering user ratings and similar users, generating collaborative recommendations, and presenting the final list of product recommendations to the user.

Here is a brief explanation of the main logic flow that the recommender system follows:

1. User Interaction and Initialization: The user starts by interacting with the system and selects the option to get a recommendation.

2. Input from the User: The system prompts the user for input, including the username, price range and preferred product category.

3. Recommendation Generation: The `recommend/5` predicate is invoked with the user's input parameters, so the system proceeds with the recommendation process based on the user's input parameters.

4. Finding Matching Products: The system identifies products that match the user's criteria using the `find_matching_products/5` predicate and filters them based on the user's preferences and criteria.

5. User Ratings and Similar Users: The system identifies similar users based on their past ratings and preferences using the `find_similar_users/3` predicate.

6. Collaborative Recommendations: Collaborative recommendations are generated using the `generate_collaborative_recommendations/2` predicate. Recommendations are based on what similar users have interacted with positively.

7. Merging and Sorting Recommendations: The system combines the matching products and collaborative recommendations, removes duplicates, and sorts the recommendations. This is done using the `merge_and_sort_recommendations/3` predicate.

8. Formatting Recommendations: The recommendations are formatted into a user-friendly list of strings and presents the recommendations to the user, showing a list of recommended products that match their criteria and preferences.

Technologies used

The following technologies were used to develop this software:

- **Visual Studio Code**: Visual Studio Code (VS Code) is a free, open-source code editor for software development with a wide range of features and extensions.
- **Prolog Language support extension**: to ease code development.
- **Github**: GitHub is a web-based platform for version control and collaboration on software development projects. It allows developers to track changes to their code, collaborate with others, and manage code repositories efficiently.

Implementation details

In this section we are going to dive deep in the code and explain how it works.

Facts

A prolog fact is a simple, declarative statement used to represent basic knowledge about a domain. It consists of a predicate with no arguments and serves as the foundational building block for defining information in Prolog programs. Facts state whether a specific proposition is true within the program's knowledge base.

The code provides three types of dynamic facts so that their content can be asserted and retracted during the execution of the program. The facts declared as dynamic are: “**category**”, “**price**”, “**user_rating**”, while all the other facts are non dynamic and cannot be changed during execution; the non dynamic facts are: “**user**” and “**search_history**”.

As stated above, there are 5 main types of facts in this recommender system:

User

The "user" fact is used to represent the **registered users** of the recommender system. It helps identify and manage users within the system. It states that there exists a user with a specific username (UserName).

The "user" fact is defined as follows: <user(UserName).>

Code snippet:

```
% users registered in the recommendation system
user(simone).
user(francesco).
user(nicholas).
user(giacomo).
```

These lines indicate that there are four users in the system with the usernames "simone," "francesco," "nicholas," and "giacomo."

Search_history

The "search_history" fact is used to represent the **search history of registered users** in a recommender system. It states that for a specific user (`User`), there exists a list of items representing their search history (`SearchHistory`). Each user's search history is represented as a list of product models or components they have previously searched for or interacted with on the platform.

The fact is defined as follows: <search_history(User, SearchHistory).>

Code snippet:

```
12  % facts about search history of registered users
13  % simone is a user who searched mostly cpus
14  search_history(simone, [intel_i9, intel_i7, amd_ryzen_7, intel_i7, intel_i5, amd_rx_6500, nvidia_rtx_3060, intel_celeron]).
15
16  % francesco is a user who searched mostly gpus
17  search_history(francesco, [nvidia_rtx_3090, amd_rx_6900, nvidia_rtx_3080, nvidia_rtx_3070, amd_rx_6800, g_skill_32gb, crucial_32gb, intel_i5]).
18
19  % nicholas is a user who searched mostly ram modules
20  search_history(nicholas, [corsair_16gb, g_skill_32gb, crucial_32gb, patriot_16gb, a_data_32gb, intel_i5, intel_i7]).
21
22  % giacomo is a user who haven't searched for nothing but still gets some suggestions
23  search_history(giacomo,[]).
```

The line 14 indicates that the user "simone" has searched for a series of products, including CPUs (e.g., "intel_i9," "intel_i7"), GPUs (e.g., "nvidia_rtx_3060," "amd_rx_6500"), ecc…

Category

The "category" fact is used to represent the **category or type of electronic components** or products. It helps categorize and organize products within the recommender system. It states that a specific product (Product) belongs to a particular category (Category).

The "category" fact is defined as follows: <category(Product, Category).>

Code snippet:

```
%cpus
category(intel_i9, cpu).
category(amd_ryzen_9, cpu).
category(intel_i7, cpu).
```

These lines indicate that: intel_i9, amd_ryzen_9 and intel_i7 belongs to the “cpu” category.

Price

The "price" facts are used to represent the **prices of various electronic components** or products within the recommender system. These facts help associate specific products with their respective prices. Each "price" fact states that a specific product (Product) has a particular price (Price).

The "price" facts are defined as follows: < price(Product, Price).>

Code snippet:

```
% Price facts
price(intel_i9, 499.99).
price(amd_ryzen_9, 749.99).
price(intel_i7, 399.99).
```

These lines indicate that the intel_i9 is priced at 499.99 €, the amd_ryzen_9 at 749.99 € and the intel_i7 is priced at 399.99 €.

User_rating

The "user_rating" facts are used to represent **user ratings or reviews for specific electronic components** or products within the recommender system. These facts help capture user feedback and opinions about products. Each "user_rating" fact states that a specific user (User) has provided a rating (Rating) for a particular product (Product).

The "user_rating" facts are defined as follows: < user_rating(User, Product, Rating).>

Code snippet:

```
% User ratings for products
user_rating(simone, intel_i9, 4.5).
user_rating(simone, amd_ryzen_9, 4.8).
user_rating(simone, intel_i7, 4.4).
```

These lines indicate that the user “simone” gave to intel_i9 a rating of 4.5/5, to the amd_ryzen_9 a rating of 4.8/5 and to the intel_i7 a rating of 4.4/5.

In summary, we can say that:

- The “category” facts are used to retrieve products that belong to the desired category.
- The “price” facts are used to filter and retrieve products that fall within the specified price range.
- The “user_rating” facts are used to identify products that a user has rated positively and recommend similar products or products from the same category.

Rules

In Prolog, a rule is a fundamental component of the language's logic programming paradigm. It defines a relationship or condition between various entities or predicates. A Prolog rule consists of two main parts: the head and the body.

The general structure of a rule follows this syntax: <Head :- Body>

Pearson_correlation

The "pearson_correlation" rule calculates the **Pearson correlation coefficient between two lists of ratings.** Pearson correlation is a statistical measure of the linear relationship between two sets of data thus is a measure of similarity or correlation between the preferences of two users, and it can be used in collaborative filtering-based recommendation systems to find similar users or recommend products based on user similarities.Rule details:

1. **Input Parameters**:
   - `RatingsUserA` and `RatingsUserB`: These are two lists of product ratings given by two different users. They are assumed to have the same length and contain numerical ratings.

2. **Calculation Steps**:
   - It calculates the length of both rating lists, ensuring they have the same length (N) as a consistency check.
   - It calculates statistics for each user's ratings, including the sum of ratings (`Sum1` and `Sum2`) and the sum of squared ratings (`SumSquared1` and `SumSquared2`).
   - It calculates the dot product of ratings between User1 and User2, which is the sum of the products of corresponding ratings.
   - It calculates the denominator for the Pearson correlation coefficient.
   - It calculates the Pearson correlation coefficient (`Score`) using the formula: `(SumProduct / Denominator)`. This coefficient measures the linear correlation between the two rating lists. If the denominator is 0 (which indicates no variance in ratings), it sets the score to 0 to avoid division by zero.

Code snippet:

```
pearson_correlation(RatingsUserA, RatingsUserB, Score) :-
    % Get the length of both rating lists (assuming they have the same length)
    length(RatingsUserA, N),
    length(RatingsUserB, N),
    % Calculate statistics for User1's ratings
    statistics(RatingsUserA, Sum1, SumSquared1),
    % Calculate statistics for User2's ratings
    statistics(RatingsUserB, Sum2, SumSquared2),
    % Calculate the dot product of ratings between User1 and User2
    dot_product(RatingsUserA, RatingsUserB, SumProduct),
    % Calculate the denominator for Pearson correlation coefficient
    denominator(N, Sum1, SumSquared1, Denominator1),
    denominator(N, Sum2, SumSquared2, Denominator2),
    denominator(Denominator1, Denominator2, N, Denominator), % Pass N as the fourth parameter
    % Calculate the Score
    (Denominator = 0 -> Score = 0 ; Score is SumProduct / Denominator).
```

All the steps mentioned above are performed using helper functions as seen in the code snippets, these helper functions are described below.

Statistics

This predicate is used within the "pearson_correlation" rule to calculate statistics for both users ratings. It is a **utility** predicate that **simplifies** the **calculation of** the **sum and sum of squares** of a list of numerical values. These statistics are used in subsequent calculations to determine the Pearson correlation coefficient between two sets of ratings.

1. **Input Parameters**:
   - `[]` and `[]`: These are the base cases for the function and represent empty lists. When both input lists are empty, the result is defined as 0.
   - `Ratings`: This parameter represents a list of numerical ratings. It is assumed to be a list of ratings given by a user for a set of products or components.

2. **Output Parameters**:
   - `Sum`: This parameter represents the sum of the ratings in the `Ratings` list.
   - `SumSquared`: This parameter represents the sum of the squares of the ratings in the `Ratings` list.

3. **Calculation Steps**:
   - The `statistics` predicate calculates the sum of the ratings (`Sum`) by summing up all the values in the `Ratings` list.
   - It also calculates the sum of the squares of the ratings (`SumSquared`) by summing up the squares of each rating in the `Ratings` list.

Code snippet:

```
% Calculate the sum and sum of squared ratings in a list
statistics([], 0, 0).
statistics([Rating|Rest], Sum, SumSquared) :-
    % Recursively calculate the sum and sum of squared ratings
    statistics(Rest, RestSum, RestSumSquared),
    Sum is RestSum + Rating,
    SumSquared is RestSumSquared + Rating * Rating.
```

Dot_product

This predicate is used within the "pearson_correlation" rule to **compute the dot product** (scalar product) of two lists of product ratings given by users.

1. **Input Parameters**:
   - `[]` and `[]`: These are the base cases for the function and represent empty lists. When both input lists are empty, the result is defined as 0.
   - `[Rating1|Rest1]` and `[Rating2|Rest2]`: These represent the heads (first elements) of the input lists along with their respective tails. In the context of calculating the dot product, these heads represent the current ratings being considered.

2. **Output Parameter**:
   - `DotProduct`: This parameter represents the result of the dot product calculation. It is a numerical value.

3. **Calculation Steps**:
   - In the recursive case, it calculates the dot product by taking the product of the current ratings (`Rating1 * Rating2`) and adding it to the dot product of the remaining elements (`RestDotProduct`). This recursive calculation continues until both input lists are empty.

Code snippet:

```
% Calculate the dot product of ratings between two lists
dot_product([], [], 0).
dot_product([Rating1|Rest1], [Rating2|Rest2], DotProduct) :-
    % Recursively calculate the dot product
    dot_product(Rest1, Rest2, RestDotProduct),
    DotProduct is RestDotProduct + Rating1 * Rating2.
```

Denominator

This predicate is used within the "pearson_correlation" rule for **calculating the denominator part of the Pearson correlation coefficient formula,** which is essential for quantifying the similarity or correlation between two sets of data, such as user ratings in collaborative filtering recommendation systems. It ensures that the denominator is calculated correctly and handles special cases to prevent division by zero.

1. **Input Parameters**:
   - `0`: This is a special case parameter used to handle situations where `N` (the length of the rating lists) is 0. It prevents division by zero by returning a denominator of 0.
   - `N`: This parameter represents the length of the rating lists. It is assumed to be a positive integer.
   - `Sum`: This parameter represents the sum of the ratings in the rating list.
   - `SumSquared`: This parameter represents the sum of the squares of the ratings in the rating list.

2. **Output Parameter**:
   - `Denominator`: This parameter represents the result of the denominator calculation, which is a numerical value.

3. **Calculation Steps**:
   - The "denominator" function consists of two clauses. The first clause is a special case for when `N` is 0 (i.e., an empty rating list). In this case, it sets the denominator to 0 to avoid division by zero. The cut operator (`!`) is used to prevent backtracking and ensure that this clause is selected when `N` is 0.
   - The second clause calculates the denominator using the Pearson correlation formula, which involves the length of the rating lists (`N`), the sum of the ratings (`Sum`), and the sum of the squares of the ratings (`SumSquared`). It computes the square root of `(N * SumSquared - Sum * Sum)` to obtain the denominator.

4. **Pearson Correlation Formula**: The denominator part of the Pearson correlation coefficient formula is used to normalize the covariance of the two sets of data. It measures the spread or variability of the data.

Code Snippet:

```
% Calculate the denominator for Pearson correlation coefficient
denominator(0, _, _, 0) :- !. % Avoid division by zero
denominator(N, Sum, SumSquared, Denominator) :-
    % Calculate the denominator using the Pearson formula
    Denominator is sqrt(N *x SumSquared - Sum * Sum).
```

Recommend

Like mentioned above this is the **main predicate** that keeps it all together. It generates product recommendations for a user based on their search history, preferences, and user ratings.

The rule proceeds by finding matching products that fit the user's criteria, considering user ratings, identifying similar users, generating collaborative recommendations, and finally formatting and presenting the recommendations to the user.

Input Parameters:

- `User`: It identifies the specific user for whom the recommendations are being made.
- `MinPrice` and `MaxPrice`: These parameters represent the minimum and maximum price range that the user is willing to pay for a product.
- `SelectedCategory`: This parameter represents the category of the product that the user is interested in. It can be a specific category (e.g., "cpu," "gpu," or "ram") or "skip" to indicate that the user does not have a specific category preference.

Output Parameter:

- `Recommendations`: This parameter represents the list of product recommendations generated for the user. Each recommendation is formatted as a string for display.

Steps:

- **Step 1: Get User's Search History**: The rule begins by retrieving the user's search history using the `search_history/2` fact.

- **Step 2: Find Matching Products**: The rule calls the `find_matching_products` predicate to find products that match the user's search history, category filter, and price range. This step narrows down the list of products to those that are relevant to the user's preferences.

- **Step 3: Get User's Ratings**: The rule retrieves the user's ratings for products using the `find_user_ratings` predicate. These ratings are crucial for identifying similar users and making personalized recommendations.

- **Step 4: Find Similar Users**: Based on the user's ratings, the rule calls the `find_similar_users` predicate to identify similar users. Similarity between users is determined using a correlation measure (Pearson correlation) based on their ratings.

- **Step 5: Generate Collaborative Recommendations**: The rule generates collaborative recommendations based on what similar users have searched for. It calls the `generate_collaborative_recommendations` predicate to find products that have been favored by similar users.

- **Step 6: Merge and Sort Recommendations**: The rule combines the products that match the user's search history with the collaborative recommendations. It removes duplicates and sorts the recommendations to ensure uniqueness and relevance.

- **Step 7: Format Recommendations**: Finally, the rule calls the `format_recommendations` predicate to format the unique recommendations into a list of strings for display.

Code snippet:

```
/* ============================================================================================================ */
% recommendation system core

% Define a recommend predicate that generates product recommendations for a user
recommend(User, MinPrice, MaxPrice, SelectedCategory, Recommendations) :-

    % Get the user's search history
    search_history(User, SearchHistory),

    % Find products that match the user's search history, category filter, and price range
    find_matching_products(SearchHistory, SelectedCategory, MinPrice, MaxPrice, MatchingProducts),

    % Get the user's ratings for products
    find_user_ratings(User, UserRatings),

    % Find similar users based on their ratings
    find_similar_users(User, UserRatings, SimilarUsers),

    % Generate collaborative recommendations based on what similar users have searched for
    generate_collaborative_recommendations(SimilarUsers, CollaborativeRecommendations),

    % Combine, sort, and remove duplicates from both collaborative and existing recommendations
    merge_and_sort_recommendations(MatchingProducts, CollaborativeRecommendations, UniqueRecommendations),

    % Format the unique recommendations into a list of strings for display
    format_recommendations(UniqueRecommendations, Recommendations).
```

All the steps mentioned above are performed using helper functions as seen in the code snippet, these helper functions are described below.

Find_matching_products

The "find_matching_products" predicate helps **filter and identify relevant products** for a user **based on their search history** and preferences. It is an essential part of the recommendation system, as it narrows down the list of products to be considered for recommendations.

1. **Input Parameters**:
    - `SearchHistory`: The search history of a user. It is a list of products.
    - `SelectedCategory`: This parameter represents the category of products that the user is currently interested in. It can be a specific category (e.g., "cpu," "gpu," or "ram") or "skip" to indicate that the user does not have a specific category preference.
    - `MinPrice` and `MaxPrice`: These parameters represent the minimum and maximum price range that the user is willing to pay for a product.

2. **Output Parameter**:
    - `MatchingProducts`: This parameter represents the list of products that match the user's search criteria, including category, price range, and previous search history.

3. **Calculation Steps**:
    - The "find_matching_products" predicate uses Prolog's `findall` predicate to generate a list of products that meet the specified criteria. The generated list is stored in the `MatchingProducts` parameter.
    - **For each product** in the user's `SearchHistory` (denoted as `Product`), the predicate performs **the following checks**:
        - It retrieves the category of the product using the `category/2` fact.
        - It compares the `SelectedCategory` with the actual category of the product.
        - It retrieves the price of the product using the `price/2` fact.
        - It checks if the price of the product falls within the user's price range
    - If all the conditions are met, the product is included in the `MatchingProducts` list.

Code snippet:

```prolog
% Find products that match the user's search history, category filter, and price range
find_matching_products(SearchHistory, SelectedCategory, MinPrice, MaxPrice, MatchingProducts) :-
    findall(Product, (member(Product, SearchHistory),
                      category(Product, Category),
                      (SelectedCategory = 'skip' ; SelectedCategory = Category),
                      price(Product, Price),
                      Price >= MinPrice,
                      Price =< MaxPrice), MatchingProducts).
```

Find_user_ratings

The "find_user_ratings" retrieves the user's historical ratings for various products. These ratings are used in collaborative filtering to identify similar users, make recommendations, and calculate similarity scores between users based on their preferences.

1. **Input Parameters**:
    - `User`: Parameter used to identify the specific user whose ratings we are interested in.

2. **Output Parameter**:
    - `UserRatings`: This parameter represents the list of ratings given by the user. Each element of the list is a pair consisting of a product and the corresponding rating.

3. **Calculation Steps**:
    - The "find_user_ratings" predicate uses Prolog's findall predicate to generate a list of pairs (Product-Rating) that represent the ratings given by the specified user (User).
    - For each product for which the specified User has given a rating, the predicate collects the product and its rating using the user_rating/3 fact. This fact associates a user with a product and a rating.
    - The collected pairs are stored in the UserRatings list

Code snippet:

```
% Find the user's ratings for products
find_user_ratings(User, UserRatings) :-
    findall(Rating-Product, user_rating(User, Product, Rating), UserRatings).
```

Find_similar_users

It is responsible for identifying and **finding users** who are **similar** to a given user **based on their product ratings**. It calculates the similarity between the target user and other users in the system using a correlation measure (Pearson correlation coefficient) and returns a list of similar users along with their similarity scores.

1. **Input Parameters**:
    - `User`: target user for whom we want to find similar users.
    - `UserRatings`: list of ratings given by the target user to various products. It is in the form of a list of pairs (Product-Rating).

2. **Output Parameter**:
    - `SimilarUsers`: output value which contains a list of pairs, where each pair consists of an "OtherUser" (a user similar to the target user) and their similarity score. For example, it might look like `[User1-Similarity1, User2-Similarity2, ...]`.

3. **Steps**:
    - The "find_similar_users" predicate uses Prolog's `findall` predicate to generate a list of similar users and their similarity scores.
    - **For each** potential "**OtherUser**" the predicate performs the following steps:

- It ensures that the "OtherUser" is not the same as the "User" (target user) by using the condition `OtherUser \= User`.
- It calls the `find_similar` predicate to calculate the similarity between the target user's ratings (`UserRatings`) and the ratings of the "OtherUser."
- The generated pairs (OtherUser-Similarity) are collected in the `SimilarUsers` list.

Code snippet:

```
% Find similar users based on their ratings
find_similar_users(User, UserRatings, SimilarUsers) :-
    findall(OtherUser-Similarity, (
        user(OtherUser),
        OtherUser \= User,
        find_similar(UserRatings, OtherUser, Similarity)
    ), SimilarUsers).
```

But the actual calculation of similarity is done in the `find_similar` predicate.

Find_similar

Used within the context of the "find_similar_users" predicate to find users who are similar to a target user. The similarity is calculated using a correlation measure, such as the Pearson correlation coefficient.

1. **Input Parameters**:
    - `UserRatings`: List of list of pairs (Product-Rating) representing the ratings given by a user.
    - `OtherUser`: This parameter represents the username of the user with whom we want to calculate similarity.

2. **Output Parameter**:
    - `Similarity` contains the computed similarity score between the target user and the "OtherUser" based on their product ratings. This score quantifies the degree of similarity between their preferences, with higher values indicating greater similarity.

3. **Steps**:
    - **Retrieve Other User's Ratings**: It calls the `find_user_ratings` predicate to retrieve the ratings of the "OtherUser" (i.e., the product ratings given by the other user to various products). These ratings are stored in the `OtherUserRatings` list.

    - **Calculate Similarity**:It calls the `pearson_correlation` predicate to calculate the similarity between the target user's ratings (`UserRatings`) and the ratings of the "OtherUser" (`OtherUserRatings`). The Pearson correlation coefficient is used as the similarity measure.

Code snippet:

```
% Define a predicate to find the similarity between two users based on their ratings
find_similar(UserRatings, OtherUser, Similarity) :-
    find_user_ratings(OtherUser, OtherUserRatings),
    pearson_correlation(UserRatings, OtherUserRatings, Similarity).
```

It is important to point out that the actual calculation of similarity relies on the `pearson_correlation` which we discussed before.

Generate_collaborative_recommendations

This is the predicate responsible for **generating collaborative recommendations for a user** based on the search and rating patterns of users who looks similar to him.

1. **Input Parameters**:
   - `SimilarUsers`: list of users who are similar to the target user. Generated by the `find_similar_users` predicate.

2. **Output Parameter**:
   - `CollaborativeRecommendations` contains a list of products that are recommended to the target user based on collaborative filtering. These products have been positively rated by users who are similar to the target user, suggesting that they might be of interest to the target user.

3. **Steps**:
   - **Iterate Over Similar Users**:
     - It uses Prolog's `findall` predicate to generate a list of products (`Product`) that have received high ratings from users similar to the target user.
       For each `OtherUser` in the `SimilarUsers` list, it performs the following checks:
       - It retrieves the rating (`Rating`) given by the `OtherUser` to the `Product` using the `user_rating/3` fact.
       - It checks if the `Rating` is greater than or equal to 4.0, indicating that the `OtherUser` has rated the product positively.

   - **Collect Collaborative Recommendations**:
     - Products that meet the criteria (i.e., have received high ratings from similar users) are collected in the `CollaborativeRecommendations` list.

Code Snippet:

```
% Generate collaborative recommendations based on similar users
generate_collaborative_recommendations(SimilarUsers, CollaborativeRecommendations) :-
    findall(Product, (member(OtherUser, SimilarUsers),
                      user_rating(OtherUser, Product, Rating),
                      Rating >= 4.0), CollaborativeRecommendations).
```

Merge_and_sort_recommendations, remove_duplicates, format_recommendations, format_product

These predicates collectively contribute to the recommendation system's functionality, ensuring that recommendations are presented in an organized, user-friendly manner, while also eliminating duplicates.

1. **`merge_and_sort_recommendations`** predicate:
   - Purpose: **Combines, sorts, and removes duplicates from two lists of product recommendations**, one based on user search history and the other on collaborative filtering.
   - Input Parameters: `MatchingProducts` (search history recommendations), `CollaborativeRecommendations` (collaborative recommendations).
   - Output Parameter: `UniqueRecommendations` (final unique recommendations). Provides a unified, sorted, and unique set of product recommendations for the user.
   - Steps:
     - Merges the two recommendation lists into `AllRecommendations`.
     - Sorts `AllRecommendations` for consistency.
     - Removes duplicate recommendations to create `UniqueRecommendations`.

2. **`remove_duplicates`** predicate:
   - Purpose: **Removes duplicate elements from a list.**
   - Input Parameter: Input list containing potential duplicates.
   - Output Parameter: Output list with duplicates removed.
   - Steps:
     - Recursively processes the input list and filters out duplicates.

3. **`format_recommendations`** predicate:
   - Purpose: **Formats a list of product recommendations into a user-friendly display format.**
   - Input Parameters: Input list of product recommendations.
   - Output Parameter: Formatted list of recommendations ready to be displayed.
   - Steps:
     - Recursively formats each product recommendation using `format_product`.

4. **`format_product`** predicate:
   - Purpose: **Formats an individual product for display.**
   - Input Parameter: Product information (e.g., name, category, price).
   - Output Parameter: Formatted product string for display.
   - Steps:
     - Constructs a string with product details, including name, category, and price.

Code snippet:

```
% Merge, sort, and remove duplicates from recommendations
merge_and_sort_recommendations(MatchingProducts, CollaborativeRecommendations, UniqueRecommendations) :-
    append(MatchingProducts, CollaborativeRecommendations, AllRecommendations),
    sort(AllRecommendations, SortedRecommendations),
    remove_duplicates(SortedRecommendations, UniqueRecommendations).

% Define a predicate to remove duplicates from a list
remove_duplicates([], []).
remove_duplicates([X|Xs], [X|Ys]) :-
    remove_duplicates(Xs, Ys),
    \+ member(X, Ys).

% Format recommendations into a list of strings
format_recommendations([], []).
format_recommendations([Product|Rest], [Formatted|FormattedRest]) :-
    format_product(Product, Formatted),
    format_recommendations(Rest, FormattedRest).

% Format a product for display
format_product(Product, Formatted) :-
    category(Product, Category),
    price(Product, Price),
    atomic_list_concat([Product, ' (Category: ', Category, ', Price: $', Price, ')'], Formatted).
```

Init and Menu_switch

The `init` rule and related procedures `menu_switch/1` form the main menu and navigation system in the recommendation system. They **allow the user to interact with the system**, make choices, and perform various actions. Here's an explanation of each component:

1. **`init`** rule:
    - Purpose: The `init` rule serves as the **entry point** to the recommendation system. It presents the main menu options to the user, allowing them to choose between user and admin functionalities or exit the program.
    - Key Actions:
        - Displays the main menu options, which include navigating to the user menu, admin menu, or exiting the program.
        - Reads the user's input choice.
        - Calls the `menu_switch/1` predicate to handle the chosen option.
2. **`menu_switch`** predicate:
    - Purpose: The `menu_switch/1` predicate is responsible for handling user input and **directing the program flow** based on the selected menu option.
    - Input Parameter: The user's menu choice, represented as a numeric input (1 for user menu, 2 for admin menu, 3 to exit).
    - Key Actions:
        - Depending on the user's choice, it calls either the `user_option/1`, `admin_option/1`, or initiates zthe exit process.
        - If the input choice is invalid, it provides an error message and prompts the user to try again.

Code snippet:

```
/* ========================================================================================= */
/* menu section */

% first function that needs to be called
init :-
    write('MAIN MENU:'), nl,
    write('1. go to user menu.'), nl,
    write('2. go to admin menu.'), nl,
    write('3. exit.'), nl,
    write('Select option...'), nl,
    read(Input),
    menu_switch(Input).

% main menu used by user
menu_switch(1) :-
    write('USER MENU '), nl,
    write('1. get suggestion.'), nl,
    write('2. review component.'), nl,
    write('3. check reviews.'), nl,
    write('4. Go back to main menu.'), nl,
    write('Select option...'), nl,
    read(Input),
    user_option(Input).

% main menu used by admin
menu_switch(2) :-
    write('ADMIN MENU'), nl,
    write('1. View all facts.'), nl,
    write('2. Insert new component.'), nl,
    write('3. Delete a fact.'), nl,
    write('4. Go back to main menu.'), nl,
    write('Select option...'), nl,
    read(Input),
    admin_option(Input).

menu_switch(3) :- write('Program exited...'),
                  halt.

% default case
menu_switch(_) :- write('Invalid operation, please try again.'),
                  nl,
                  init.
```

User_options

The `user_option/1` predicate is responsible for **handling user menu** options and directing the program flow based on the user's selected option within the user menu.

The predicate takes just one parameter, which represents the user's choice as a numeric input.

- Key Actions:
  - Depending on the user's choice, it calls one of the following user-specific functionalities:
    - 1. `get_recommendation`
    - 2. `review_component`
    - 3. `listing(user_rating)`
  - If the input choice is invalid, it provides an error message and prompts the user to try again.
  - After executing the chosen user-specific option, it returns to the user menu for further selections.

Get_recommendation

Typing “1”, triggers user_option(1) and activates the get_recommendation predicate which is responsible for providing **product recommendations** to the user based on their input and search history.

Key Actions:

- Prompts the user to enter their username.
- Checks the user's search history to determine if they have previous searches.
- If the user has no search history, it provides default recommendations.
- If the user has search history, it prompts the user to enter minimum and maximum price limits and a selected category (or allow the system to choose).
- Calls the `recommend/5` predicate (discussed above) to generate personalized recommendations based on the input the users just typed in.
- Presents the recommendations to the user for display.

Code snippets:

```
user_option(1) :- get_recommendation,
                  nl,
                  menu_switch(1).
```

```
get_recommendation :-
    write('Enter username: '), nl,
    read(User),
    (   search_history(User, SearchHistory) ->
        (   SearchHistory = [] ->  % Check if the user's search history is empty
            % User has an empty search history, recommend default products
            DefaultRecommendations = [amd_ryzen_7, nvidia_rtx_3070, g_skill_32gb],
            write('Since your search history is empty, we recommend the following products:'), nl,
            print_recommendations(DefaultRecommendations)
        ;
            write('Enter the minimum price you are willing to pay: '), nl,
            read(MinPrice),

            write('Enter the maximum price you are willing to pay: '), nl,
            read(MaxPrice),

            write('Enter the category of the product you are searching for (cpu, gpu, or ram),'), nl,
            write('or enter "skip" to let the system choose a category for you:'), nl,
            read(SelectedCategory),

            recommend(User, MinPrice, MaxPrice, SelectedCategory, Recommendations),
            nl,
            write('Recommendations for '), write(User), write(':'), nl,
            print_recommendations(Recommendations)
        )
    ;
    write('User not found.')).
```

Review_component

Typing “2” instead, triggers user_options(2) and thus activates the review_component predicate, which allows users to **review and rate a specific component** in the system.

- Key Actions:
  - Prompts the user to enter their username, the component's model, and a rating for the component.
  - Uses the `assert/1` predicate to add the user's rating to the `user_rating/3` dynamic fact.

Code snippets:

```
user_option(2) :- review_component,
                  nl,
                  menu_switch(1).
```

```
review_component :-

    write('Enter username:'), nl,
    read(Username),

    write('Enter component\'s model:'), nl,
    read(Model),

    write('Enter rating:'), nl,
    read(Rating),

    assert(user_rating(Username, Model, Rating)),
    write('Review added.'), nl.
```

Listing(user_rating)

Lastly, typing “3” activates the `listing(user_rating)` predicate allows users to view existing reviews and ratings provided by users for different components in the system.

Key Actions:

- Lists the user ratings and reviews stored in the `user_rating/3` fact, displaying the product model, user, and rating.
- This functionality provides users with the ability to see reviews provided by themselves and others.

Code snippet:

```
user_option(3) :- listing(user_rating),
                  nl,
                  menu_switch(1).
```

Admin_options

The `admin_option/1` predicate is responsible for **handling admin menu options** and directing the program flow based on the selected option within the admin menu.

The predicate takes just one parameter, which represents the user's choice as a numeric input.

- Key Actions:
    - Depending on the user's choice, it calls one of the following user-specific functionalities:
        - 1. `print_all_facts`
        - 2. `add_component`
        - 3. `delete_component`
    - If the input choice is invalid, it provides an error message and prompts the user to try again.
    - After executing the chosen user-specific option, it returns to the user menu for further selections.

Print_all_facts

Typing "1" activates the `print_all_facts` predicate which allows administrators to **view all facts in the knowledge base**, including product categories, prices, and user ratings.

Key Actions:

- Uses the `listing/1` predicate to display the facts for `category`, `price`, and `user_rating`. This provides administrators with a comprehensive view of the system's data.

Code snippets:

```
admin_option(1) :- print_all_facts,
                   nl,
                   menu_switch(2).
```

```
print_all_facts :-
    listing(category),
    listing(price),
    listing(user_rating).
```

Add_component

Typing “2” activates the add_component predicate which enables administrators to **add new components** (CPUs, GPUs, RAM modules) to the knowledge base, specifying their category and price.

Key Actions:

- Prompts the admin to enter the component's category, model, and price.
- Uses the `assert/1` predicate to add the new component's category and price to the knowledge base.

Code snippets:

```
admin_option(2) :- add_component,
                   nl,
                   menu_switch(2).
```

```
add_component :-

    write('Enter component\'s category (cpu, gpu, ram):'), nl,
    read(Category),

    write('Enter component\'s model:'), nl,
    read(Model),

    write('Enter component\'s price:'), nl,
    read(Price),

    assert(category(Model, Category)),
    assert(price(Model, Price)),

    write('Component added.'), nl.
```

Delete_component

Typing “3” activates the `delete_component` predicate, which allows administrators to remove components from the knowledge base based on the component's model.

Key Actions:

- Prompts the admin to enter the model of the component they want to delete.
- Uses the `retractall/1` predicate to remove all facts related to the specified component, including its category, price, and user ratings.

Code snippets:

```
admin_option(3) :- delete_component,
                   nl,
                   menu_switch(2).
```

```
delete_component :-

    write('Enter component model to delete from knowledge base:'), nl,
    read(Model),

    retractall(category(Model, _)),
    retractall(price(Model, _)),
    retractall(price(user_rating, _)),

    write('Component removed.'), nl.
```

Achieved results and conclusions

In conclusion, the recommender system project has achieved its objectives of providing **personalized product recommendations**, enabling user reviews and ratings, and offering an interactive and user-friendly interface. It has also empowered administrators to manage and maintain the system's knowledge base efficiently. Key highlights of the project include:

1. **Personalized Product Recommendations**: The system provides personalized product recommendations to users based on their search history, price preferences, and selected product categories. Users can receive tailored suggestions that match their interests and budget.

2. **Collaborative Recommendations**: The system incorporates collaborative filtering to provide recommendations based on similar users' preferences and reviews. This enhances the quality of recommendations and ensures a diverse set of product suggestions.

3. **User Reviews and Ratings**: Users have the ability to review and rate products or components within the system. These reviews and ratings are stored and can be viewed by other users.

4. **User-Friendly Interface**: The system offers a user-friendly interface with a menu-driven approach. Users can easily navigate through different functionalities, making it accessible to use.

5. **Administrative Capabilities**: Administrators or authorized users can manage the system's knowledge base by adding new components, deleting components, and viewing all facts. This ensures that the system remains up-to-date and adaptable to changes in product offerings.

6. **Dynamic Knowledge Base**: The system maintains a dynamic knowledge base with dynamic facts for categories, prices, and user ratings. This flexibility allows for easy updates and modifications to the system's data.

Possible future developments and improvements

Some possible future developments and enhancements for the recommender system may include:

1. **User Authentication**: Implement user authentication and user account management to ensure effective user-specific data privacy and security.

2. **More efficient code**: improve the code by providing solid mechanisms to check the syntax and manage the errors more efficiently.

3. **Multi-Criteria Recommendations**: Allow users to specify multiple criteria for recommendations, such as performance, price, and brand, and provide recommendations that optimize across these criteria.