

The Hong Kong Polytechnic University
Department of Computing

COMP4913 Capstone Project
Report (Final)

**Personalized Search and
Recommendation Services**

Student Name:	Chang Wai Chung
Student ID No.:	21052435D
Programme-Stream Code:	61431-SYC
Supervisor:	Prof. LI Qing
Co-Examiner:	Dr HUANG Xiao
2 nd Assessor:	Dr WU Xiaoming
Submission Date:	11 April 2023

Abstract

This report proposes Recommendation System (RS) and Personalized Search System (PSS) for an online store that is still in-development and has limited of dataset. The objective of this report is to provide methods to build RS and PSS for the online store. Building a temporary online store is necessary as it will be used for manually generating dataset and giving a foundation for the development of RS and PSS.

RS and PSS will need to analyze user interests to be functional, so each user will have a customer profile. Customer profile calculate the weight of user's interested type of items by their rating on purchased products. Because it can find the type of product user interested the most, so the PS and RSS can accurately recommend appropriate products and personalized search results.

RS use hybrid filtering which is consist of Content-Based Filtering (CBF) and Collaborative Filtering (CF). CBF is to analyses the content of product, and CF is to find similar user. CBF utilize the customer profile to find out the possible interested weight of products, and CF input the customer profile of users to k-Nearest Neighbors machine learning models to find similar users.

The RS and PSS was successfully implemented and built by using the strategy discussed in methodology, it proved that even without a real-life dataset, building a RS and PSS for small online business to help them generate sales is possible.

Table of Contents

Abstract.....	2
Tables List.....	5
Introduction	6
Literature Review	7
Problem Statement.....	10
Objective	11
Methodology.....	12
Temporary Online Store.....	12
Products	13
Recommendation System	13
Customer Profile	14
Content-Based Filtering Recommendation.....	16
Collaborative Filtering Recommendation	17
Basic Searching System	17
Personalized Search System.....	18
Database	20
Implementation	21
Programming language and Framework.....	21
Online Store	21
Pages	22
Database	26
Basic Search System	27
Building Customer Profile	30
Content-Based Filtering Recommendation System	33
Finding Similar Users.....	37
Collaborative Filtering Recommendation System.....	38
Personalized Search System.....	39
Evaluation	41
Result	41
Product Recommendation	41
Personalized Searching	45
Recommendation System Drawbacks	46

Discussion..... 48

 Large Dataset Expansion Solution..... 48

Conclusion..... 49

Reference 50

Tables List

Table 1: Rating Example.....	14
Table 2: Sum of Rating	15
Table 3: Customer Profile Example	15
Table 4: Customer Profile Example for New User	15
Table 5: Sorted Customer Profile Example	16
Table 6: Database Tables.....	20

Introduction

The popularity of digital products has been significantly increased by the growth of e-commerce. This is because of the immediate accessibility of digital products and the efficiency of online shopping [1]. This has created incentive for artists to monetize their digital creations and establishing online stores.

Recommendation is about providing a good option for people to choose. It is effective and useful when it comes to a situation where user is overloaded with the large amount information and options [2]. Like the job of a salesman, recommending items after studied the target's needs and interests, In the digital world, this concept is exact the same as Recommendation system, collecting data about the users, understand their interests, and make recommendations.

Now, E-commerce platforms with Recommendation system (RS) and Personalized search system (PSS) have further enhanced the benefits. With the help of RS and PSS, 3D digital artists can market and promote their products more effectively and reaching new customers who may not have been previously aware of their work and increasing sales potential [3]. RS and PSS could be very useful for 3D digital artists to reach a wider audience and increase their income potential.

RS is a filtering system that can recommend user's possible interested items by analyzing their data, such as preference, purchased histories, browsing histories, or feedbacks etc. Each user will have their own profile created by RS [4]. The more the user interact with the application, the more precious of the user profile could be, and so the higher chance to recommend an appropriate new item to the targeted user. PSS is like RS, but it instead provides a personalized searching results to users by re-ranking the results. The goal of the system is to prioritize relevant search results to user base on their preference or interests, so user profile will be used as well.

This project is about building a RS and PSS for a 3D model online store that is still under development and yet to contain any real-life dataset, especially for small online business. This report will discuss and explain how this project can be achieved.

Literature Review

Recommendation system for E-commerce have been well researched.

This paper [5] was written to analyze methods of building a RS using a large Netflix dataset that consists of 17K movies and 500K+ customers, and then discover the most suitable methods out of all option for that enormous dataset. It has introduced and explained four types of worth-mentioning recommender algorithms or approaches:

Popularity, Collaborative Filtering, Content-based Filtering, and Hybrid Filtering.

- **Popularity** is a straightforward strategy, which is recommending the most popular items to clients, those recommended popular item could be the most purchased, view or highest rated in the platform. This strategy is suitable for new client who have not rated or view any items yet. However, it lacks flexibility and expansion, because it is not possible for new items to be recommended and it is difficult to provide opportunities to learn more about both client and items to improve the platform.
- **Content-Based Filtering** is a recommendation algorithm that recommend similar items that the client have viewed, purchased, or liked. Similar means the similar information of the recommended product, such as type, creator, and genre etc., a possess information to describe each item is needed, and one of the best information is textural descriptions. To recommend product to client, the algorithm need to learn the preference of each client to recommend items that are similar to their preference. The paper has mentioned two difficulties: “Limited content analysis” and “content overspecialization”.
 1. **Limited content analysis** happens frequently when RS depends on descriptive information, item publisher might use popular vocabulary to gain higher chance of being recommended.
 2. **Content overspecialization** means that client could be recommended to the entire collection of a series, for example all Lord the rings movies.

- **Collaborative Filtering**'s main objective is to find similar clients to a targeted client. Similar clients means that they could have similar interested and preference as the targeted client. After discovered similar clients, this algorithm will recommend similar client purchased products to the targeted client. However, it has few problems:
 1. **Cold start**, which means new items and clients are unable to be involved, because they have provided insufficient information to the system, so new items cannot be able to recommend to clients, and new clients cannot be served by RS.
 2. **Sparsity**, it happens to unpopular items with only few ratings, those items will be very difficult to be recommended to clients if the platform are full of popular items.
 3. **Scalability**, since CF uses item-client matrix to find similar clients, the more of the items and clients in the platform, the more of computation power for the larger matrix, which could negatively-impact the RS performance.
 4. **Popularity based**, sometimes CF could always recommend the most popular items in the platform to targeted client as there are lots of clients have highly rated the same popular items.
- **Hybrid Filtering** combines recommendation approaches not only to increase recommendation diversity, but also to solve all their own problems to each other. There is no restricted rule on how it should be done. For example, one of the common methods is using client profile that is created by CB to find similar clients by CF. Therefore, using Hybrid Filtering is more reliable and stable than using only one recommendation approaches. It is considered as hybrid filtering approaches if the system uses more than one recommendation approaches for RS.

Next, this paper [2] is about building a RS with insufficient data, it indicated that it is possible to build a RS for a newly established online store, implying any scale of online business always have an opportunity to create a RS. Some of the publicly available

recommendation approaches the researchers evaluated such as content-based, collaborative filtering and hybrid filtering were being considered as the “beginner” approaches for new platform with insufficient data.

There are two new existing problems that were not mentioned in previous paper are: Self-biased and Popularity bias.

1. **Self-biased**, it means that RS keep recommending items with same genre to the clients. Overloading the entire recommendation list with the same type of items. Like the content-based’s problem of **content overspecialization**, the difference is RS with self-biased issue is unable to recommend other genre of items.
2. **Popularity bias**, this is an issue of CF, some clients could have some unique special taste that is very different to most of the clients, it is problematic for the system to recommend items to that special client. This can be solved by CB.

Based on the two research, the algorithm or strategies that are used RS are also suitable for PSS. Because the fundamental idea of both systems is to provide the most relevant items to clients, and those items fits clients’ needs, interests, and preferences. So, it is appropriate to integrate the idea of RS to PSS. It will be discussed in detail in the “methodology” section. Overall, this literature review can be seen as ideas of implementing a RS and PSS to new online store.

Problem Statement

In pervious section, all publicly available recommendation system approaches were mentioned, each of them has their own drawbacks.

Any small online business or beginner might not be able to build a RS as advanced as those that are built by large cooperations like Instagram or Facebook etc . as they are able to collect a large amount of data anytime to analyze and optimize their RS.

The main problems the project will encounter and needs to be investigating is:

1. How to build RS and PSS without real-life data of the online store?
2. How to analyze user's interests?
3. How will PSS work with RS?

Those three statements revolve around the common difficulties when building a RS and PSS for a new online store. Giving solutions to these difficulties can allow us to establish a solid ground for creating a new RS and PSS.

Objective

In a broad sense, the main objective is creating a RS and PSS for 3D Model online store that is still under-development and does not contain data yet. However, we can further conclude the objectives would be:

1. Building a RS that can pick relevant products to target user.
2. Building a PSS with the help of RS to prioritize relevant searching results to user.
3. Making RS and PSS functional with self-made dataset.

Overall, the objectives address the problems of building RS and PSS for a newly established online store. Achieving these objectives will also prove that any beginner online business can build a functional RS and PSS anytime.

Methodology

Temporary Online Store

Before building a RS and PSS, it needs to an online store, the purpose of the online store in this project is to implement RS and PSS. However, building an entire online store is not necessary, we could only include some of the basic and necessary features of an online store to build a temporary one:

1. Adding new users
2. Adding new products
3. Searching products
4. Purchasing products
5. Rating purchased products.
6. Creating purchase history

Because it is a temporary online store that is only for the implementation of RS and PSS. It is not necessary to implement authentication, security system, or checkout system etc.

And it should allow developer to build a RS and PSS effectively. Therefore:

1. Login system and checkout system will be excluded. Because data that are created by these two systems does not provide any value for the development of RS and PSS, and they are outside the context of RS and PSS.
2. The role of administrators is assigned to all users. Roles in this temporary online store is not important as there will be no client in this online store yet. Additionally, it will be more effective for developer to generate new dataset to build and test RS and PSS.
3. The user-interface should be designed to make developer generate new data and build RS and PSS effectively and conveniently, the developer will not constantly travel a lot of pages just to add data to dataset. So that the RS and PSS system can later have data for testing results.

The online store should also allow users to input new user' favorite product type and subtype by ticking checkboxes to act like a survey, so that recommendations can be made for new user before he/she purchased anything.

Products

Since the RS and PSS are built for a 3D model online store. So, all products are digital goods and have their only type and sub-type(s).

Type: 3D Print, Game, PBR.

Sub-type: Animal, Character, Food, Plant, Weapon, Vehicle.

The rules for the products are:

1. Products can only be purchased once.
2. Products' ratings can be changed at any time.
3. Each product can only have one type but can have multiple or none sub-type.

Recommendation System

In Literature Review section, four RS approaches were mentioned: Popularity, Collaborative Filtering, Content-based Filtering, and Hybrid Filtering.

Popularity might not be a good option for the store in this case, because the purpose of recommendation is to provide good relevant options to clients. Picking a product based on their highest in some of the aspects might not effectively attract clients and might not be useful enough for the customer to find their interested products.

Also, popularity provides no exploration for the store to observe the patterns of relationship between the users and products, so it does not have or have very little of the innovation potential. [5] Therefore, it might not be worthy to be developed for the opportunity of this project.

Content-Based Filtering (CBF) would be a good option for most of the online store because it analyses the content of products and target user to make recommendations accordingly. In this case, the RS should analyze what kind of product type the targeted user is interested in by observing their purchased histories, and then create a customer profile for the targeted user. Therefore, it is suitable for this online store.

Collaborative Filtering (CF) further expand the exploration for targeted user by finding their similar user, which means recommending products that users may not have considered or thought of before, it can offer novelty to targeted user, which is something

CBF cannot do [6]. However, it has cold start problem, items that have never been purchased can't be recommended.

So, in this case, RS should adopt CBF with CF, because CBF can recommend new products, which can solve the cold-start problem, which could also alleviate sparsity problem because CBF will not make popular item to be recommended frequently.

And, to find similar users, CF can use customer profile that is created by CBF. This could solve scalability problem for CF, because it will not need to create a large item-client matrix to find similar user.

Therefore, the RS will adopt CBF and CF hybrid approach, which means CF will rely on CBF to generate customer profile. And there will be two different recommendation list in the store, and they are separately generated by CBF and CF, so that I can differentiate the results made by CBF and CF.

Customer Profile

Customer profile records the weight of user interests in each type of products by analyzing the ratings of their purchased products. It is the most crucial part of both RS and PSS system. It can tell user's interests on certain types of products. Therefore, customer profile must be created or updated first before making any recommendations.

To calculate the weights of interest, all user's rating on products will need be taken. Giving some of a visualization will explain the calculation better:

Assume customer A had purchased three 3D models:

Customer A Rating

Item	Rate	Type	Sub Type
A	5	Game	Weapon
B	1	PBR	Animal
C	4	Game	Character

Table 1: Rating Example

Then, turning it into a matrix that sum rating for each 3D model type column:

Sum of Rating

	3D Print	Game	PBR		Animal	Char.	Food	Plant	Weapon	Vehicle
Sum	0	5+4 = 9	1		1	4	0	0	5	0

Table 2: Sum of Rating

Lastly, we can normalize the rating matrix to create customer profile by Normalization function in coding. However, the normalization is done separately for 3D model's type and subtype. Normalization is necessary when comparing with other customer profiles so that it the number of purchased products of customer does not affect finding similar users.

Customer A Profile

	3D Print	Game	PBR		Animal	Char.	Food	Plant	Weapon	Vehicle
Weight	0	0.993884	0.110432		0.154303	0.617213	0	0	0.771517	0

Table 3: Customer Profile Example

The reason model type and sub-type needs to be separated is that they can coexist together. For example, a 3D printable animal model, a plant 3D game model etc.

Customer Profile can also be made without purchasing any products. This case only applies to new users who have provided their interested type and subtype of product.

Assume a new user inputted interested type is Game, 3D Print, Food, Plant and Vehicle, the customer profile can be created directly by fill the corresponding column with 1:

New Customer Profile

3D Print	Game	PBR		Animal	Char.	Food	Plant	Weapon	Vehicle
1	1	0		0	0	1	1	0	1

Table 4: Customer Profile Example for New User

Normalization might not work since they are only filled with 1 and 0, normalized value for each column would simply be the original value itself. Additionally, each column should maintain the values as binary because the input is done by checkboxes, implying there is no "levels" exist, so 1 indicates interested, 0 indicates not interested.

Content-Based Filtering Recommendation

CBF will recommend products to user based on user's interested product type and subtype of a product. And this is the first list of recommendation in the store, which will be indicated as "You might be interested in: ".

To examine which type and sub-type of product the user is interested in the most, in customer profile, we can sort the column by weight in descending order. Let's take customer A's profile from previous section as example.

Sorted Customer A Profile

	Game	PBR	3D Print	Weapon	Char.	Animal	Food	Plant	Vehicle
Weight	0.993884	0.110432	0	0.771517	0.617213	0.154303	0	0	0

Table 5: Sorted Customer Profile Example

Based on the sorted weights, the RS will recommend a 3D model that is Game, Weapon and Character 3D models to customer A more.

A list of recommended products can be separated into two parts:

1. The first part is to pick products based on the highest weights on type and subtype, which is called prioritized recommendation. In this case, the RS will pick products that are Game Weapon 3D model, because Game and Weapon has the highest weight in the customer profile. The benefit of this approach is that it can ensure the RS will pick the type of 3D model the user interested the most.
2. The second part is picking products by calculating the weights on products, which is called highest weight recommendation. The RS will take unpurchased products based on the type of products that the user has purchased, then adding weights to those products, then pick and recommend few of those with highest weight to user.

In this case, the RS will firstly pick all of product that is a Game, PBR, Weapon, Character, or Animal 3D models. After that, the RS will add and sum the weights of the product. For example, the weight of a Game Animal Character 3D model would be: $0.993884 + 0.154303 + 0.617213 = 1.7654$. Then, the RS will pick the top few products with highest weight.

After the two separated parts are finished, the two different recommendation list will be combined and will be a result for the CBF recommendation.

However, sometimes the customer profile is based on the inputted interested type, meaning it is a new user who did not purchased anything yet. To make CBF recommendation for them, the RS can simply pick all the products based on the type of products that the user is interested in, and then randomly pick few of those products and recommend them to user.

Collaborative Filtering Recommendation

Products that are picked by CF will be the second list of recommendation in the store, which will be indicated as “Similar user have also bought those: ”

CF will pick similar users and recommend their highly rated products that are not acquired by target users. Therefore, the most crucial part of CF is the method of determining similar users, and the critique of defining “similar”.

CF will utilize customer profile to find similar user, but it needs to exclude users who have not purchased anything.

To find similar users, CF will use **k-Nearest Neighbors (kNN)**. **kNN** is a supervised Learning machine learning model to find nearest neighbors of a targeted datapoint, which can be used for RS [7]. The targeted datapoint is the targeted user, and the nearest neighbors are the similar users. The kNN will also be suitable for customer profile where its weight only consists of 1 and 0.

The reason kNN was picked instead of other machine learning model is that it does not need to train, implying it can still search the most similar user regardless of how much input data it has.

After finding similar users, the RS will recommend products that were purchased by the similar users to targeted user.

Basic Searching System

Before implementing PSS, the online store needs a basic searching system.

It is a Boolean-based search system, which will take all the product that partly or fully matches the query words by words. And the search results can also be filtered by product's type or subtype. Explaining it in example would be easier to comprehend.

Assume a user wants to find an 3D printable Iron Man gauntlet 3D model. So, he inputted "Iron Man Gauntlet" to search bar, and only tick "3D Print" type checkbox in the filter list.

Firstly, the system will take the "Iron Man Gauntlet" query from user, and then break it down into tokens, which will be "Iron", "Man", and "Gauntlet".

Secondly, the system will take all the products whose name contains one of or all tokens, into a temporary list.

Lastly, the system will delete all models that is not 3D Print type from the temporary list.

And that's the functionality of basic search system.

Additionally, there is a simply logical expression for the search type filtering:

- Type **or** Type
- Subtype **or** Subtype
- Type **and** (Subtype **or** Subtype)

For example, if a user is looking for a 3D printable, Game, Animal and weapon models, the system will give 3D printable or Game models that are also Animal or Weapon models.

Personalized Search System

Personalized Search System will use on customer profile to re-sort or re-rank products inside a searching result list that is created by basic search. The way it works is very similar to Content-Based Filtering because it will also use customer profile to assign weights to all products inside the searching result list. The higher the weight of a product, the higher of prioritization in the new sorted result list.

However, sometimes a product that only fulfilled one token and gained highest weight can be prioritized to the most top of the list. For example, the search query is Iron Man Gauntlet", but there is a product named "Gauntlet" is somehow become the 1st product in the search result list.

This situation would reduce the relevance or reliability of the searching system.

To avoid this situation, PSS needs to divide the search result list into three parts in order: “All fulfilled”, “Query fulfilled” and “Partially fulfilled”.

“All fulfilled” means the name of the product completely matches with the query, and the length of the name also matches the length of the query.

“Query fulfilled” means the name of the products can be all fulfilled with the query.

“Partially fulfilled” means the name of the products only partially fulfilled with the query.

For example, there is a search query is “Iron Man”. A product named “Iron Man” is considered as “All fulfilled”, another one named with “Iron Man Helmet” is considered as “Query fulfilled”, and the last one that named with “Iron Ingot” is considered as “Partially fulfilled”. So, with this approach, even though “Iron Ingot” has the highest weight among all the items, it will still not be at very top of the list of searching result.

In short, basic search system will create a list of products that fulfilled the query, then PSS will use customer profile to assign weight to the products and sort all of them inside the list, but all the products will be sorted not only based on customer profile, but also the relevance to the search query.

Database

Database is used for storing every information and data of the store.

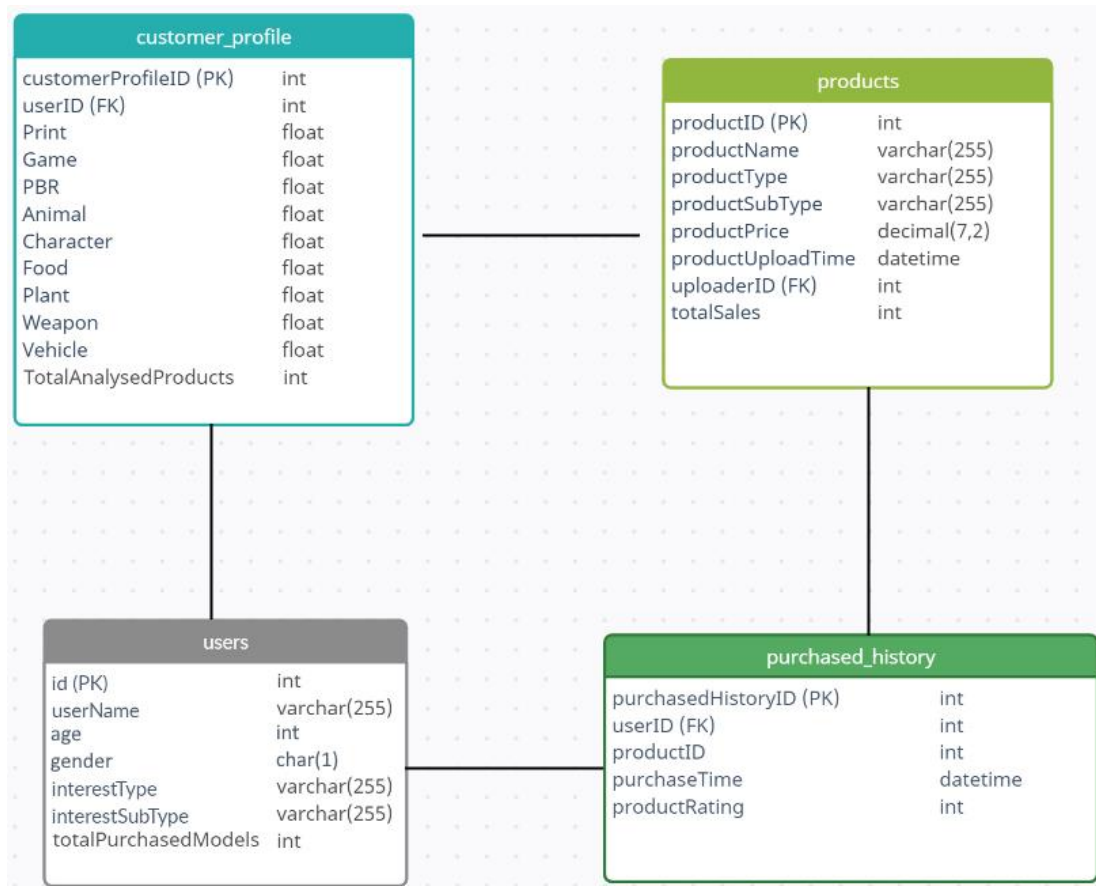


Table 6: Database Tables

RS and PSS will only need four tables to be functional.

- “users” is to store the information of a user, “interestType” and “interestSubType” can be empty.
- “product” is to store all the information of a product.
- “purchased_history” is to store user’s purchase records and ratings on products.
- “customer_profile” is to store the weight of user’s interest to product types.

Implementation

This section will mainly describe the important part of the code of the online store and RS. The code will be explained in steps for readability.

Programming language and Framework

The online store will be built by Flask, which is a Python web framework. The reason to use Python language is that it has also contain a library for implementing kNN to RS CF easily.

Bootstrap v5 was also used, to make the layout much easier to read.

Online Store

The functionality of the online store consists of three python files: **app.py**, **dbmodel.py**, **RS.py**, **PSS.py**.

- **app.py**: This file is the core of the store. It defines routes for the different pages of the store, and it has functions that handle the user requests and responses, such as adding users, purchasing, and rating products etc. It also defines the templates and static files that are used to display the web pages.
- **RS.py**: It contains the code of entire RS. Building customer profiles, finding similar users, recommending products with CBF and CF, these functionalities are all implemented in this file.
- **PSS.py**: It contains the code of PSS which is responsible for sorting search results based on customer profiles.
- **dbmodel.py**: It creates and define tables for the database. It will also allow other python code to create new data to database.

The following code at the beginning of **app.py** will set up the online store:

```
app = Flask(__name__)

#####
app.config['SECRET_KEY'] = 'FDHGFHGH FHRTGHDFHRTHRH'

#localhost
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql+pymysql://root:123456@localhost/fyp'

db = SQLAlchemy(app) #use SQLAlchemy to connect mySQL
from dbmodel import Users, Products, PurchaseHistory, CustomerProfile #import classes/tables, so must be placed before db.create_all()

with app.app_context():

    if(Users.query.filter_by(userName="DU").first() is None):
        newUser = Users(userName = "DU")
        db.session.add(newUser)
        db.session.commit() #Confirm and update

    #####
login_manager = LoginManager()
login_manager.login_view = 'app.userProfilePage_User'
login_manager.init_app(app)

@login_manager.user_loader
def load_user(userID):
    return Users.query.get(userID)
```

The app will automatically create and login to a new user called “DU”. Since the website does not need any login system.

LoginManger is initializes and configures the Flask-Login extension. Despite we don’t need login system, it can still be used for switching users, and can make the same user is still using the same user account after the store is down.

Pages

The online store has 5 pages in total: Home, Search, User Manager, Product, and Purchase. User can search products in every pages.

1. **Home** page is to show recommendation list of CF and CBF directly.

COMP Home 10/2

You might be interested in:			
ID	Name	Type	Sub-Type
3	Toy Story - Woody	Game	Character
11	Mr Bean	Game	Character
15	Monke	Game	Animal Character
66	Evangellon Shingji	Game	Character
21	Ghost	Game	Character
28	DoomSlayer	Game	Character
31	Joker	Game	Character
43	Master Chief	Game	Character
58	Iron Man Mark 42	Game	Character
56	Halo Cortana Game Model	Game	Character

Similar user have also bought those:					
Similar User	User Rating	Product ID	Product Name	Type	Sub-Type
4	5	13	Rifle	Game	Weapon
2	5	28	DoomSlayer	Game	Character
2	5	12	Pistol	3DPrint	Weapon

2. **Search** page is to display the results of searching.

COMP Home 10/2

Search Result			
ID	Name	Type	Sub-Type
67	Demon Slayer Tanjiro's Sword	Game	Weapon
40	Pirate Sword	3DPrint	Weapon

3. **User Manager** page is to manage all users, switch users and view current user's customer profile.

Add User

☐ Male ☐ Female

Interest

Type: ☐ 3D Print ☐ Game ☐ PBR

Sub-Type Model: ☐ Animal ☐ Character ☐ Food ☐ Plant ☐ Weapon ☐ Vehicle

Add

Switch User

Click

Change Interest

Interest

Type: ☐ 3D Print ☐ Game ☐ PBR

Sub-Type Model: ☐ Animal ☐ Character ☐ Food ☐ Plant ☐ Weapon ☐ Vehicle

Change

Current User

ID3

UsernameU2

Interest Type

Interest Sub-Type

3DPrint	Game	PBR	Animal	Character	Food	Plant	Weapon	Vehicle	Rated Products
0.0	1.0	0.0	0.0	0.624695	0.0	0.780869	0.0	0.0	2

4. **Product** page is to allow user to manage products.

Add Product

Type:
☐ 3D Print
☐ Game
☐ PBR

Sub-Type Model:
☐ Animal
☐ Character
☐ Food
☐ Plant
☐ Weapon
☐ Vehicle

Add

Modify Product

Type:
☐ 3D Print
☐ Game
☐ PBR

Sub-Type Model:
☐ Animal
☐ Character
☐ Food
☐ Plant
☐ Weapon
☐ Vehicle

Change

5. **Purchase** page is to allow user to purchase and rate products, and view purchased history.

Purchased History

Show ^

ID	Rating	Type	Sub-Type	Name
10	5	Game	Plant	Sunflower
2	4	Game	Character	Iron Man

Purchase Model

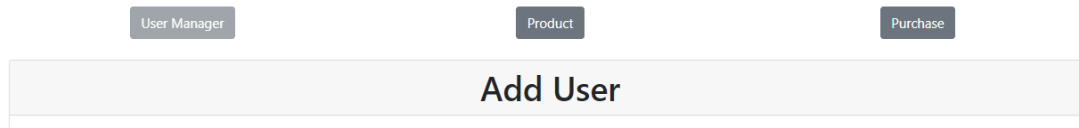
Buy

Model Rating

Rate

The layout on these pages is efficient-oriented, managing users and products and purchasing products can be done within the same page, so that the user can generate the

data a lot faster. Additionally, to travel between pages faster, buttons above pages were added:



Database

MySQL will be used for storing all data of the system. And SQLAlchemy will be used for making Flask framework connect to MySQL database.

In **dbmodel.py**, it contains code that defines four models: Users, Products, PurchaseHistory, and CustomerProfile.

```
class Users(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key = True)
    userName = db.Column(db.String(255), unique=True)
    age = db.Column(db.Integer, unique=True)
    gender = db.Column(db.String(1), unique=True) #M/F

    interestType = db.Column(db.String(255))
    interestSubType = db.Column(db.String(255))
    totalPurchasedModels = db.Column(db.Integer)

    purchasedModels = db.relationship('PurchaseHistory')
    uploadedModels = db.relationship('Products')

class Products(db.Model):
    productID = db.Column(db.Integer, primary_key = True)
    productName = db.Column(db.String(255))
    productType = db.Column(db.String(255))
    productSubType = db.Column(db.String(255))
    productPrice = db.Column(db.Numeric(7,2))

    productUploadTime = db.Column(db.DateTime(timezone=True), default=func.now())
    uploaderID = db.Column(db.Integer, db.ForeignKey('users.id'))

    totalSales = db.Column(db.Integer)

class PurchaseHistory(db.Model):
    purchaseHistoryID = db.Column(db.Integer, primary_key = True)
    userID = db.Column(db.Integer, db.ForeignKey('users.id'))
    productID = db.Column(db.Integer, db.ForeignKey('products.productID'))
    purchaseTime = db.Column(db.DateTime(timezone=True), default=func.now())
    productRating = db.Column(db.Integer)

class CustomerProfile(db.Model):
    customerProfileID = db.Column(db.Integer, primary_key = True)
    userID = db.Column(db.Integer, db.ForeignKey('users.id'))

    Print = db.Column(db.Float)
    Game = db.Column(db.Float)
    PBR = db.Column(db.Float)
    Animal = db.Column(db.Float)
    Character = db.Column(db.Float)
    Food = db.Column(db.Float)
    Plant = db.Column(db.Float)
    Weapon = db.Column(db.Float)
    Vehicle = db.Column(db.Float)

    TotalAnalysedProducts = db.Column(db.Integer)
```

Basic Search System

The system is implemented in app.py.

Type		Sub-Type
<input type="checkbox"/> 3D Print	AND	<input type="checkbox"/> Animal
<input type="checkbox"/> Game		<input type="checkbox"/> Character
<input type="checkbox"/> PBR	OR	<input type="checkbox"/> Food
		<input type="checkbox"/> Plant
		<input type="checkbox"/> Weapon
		<input type="checkbox"/> Vehicle

This is a filter list that can allow user to find specific type and sub-type of products.

First, take the searching query and the requested types from searching box and split them into array:

```
urlSearchBar = request.args.get('searchBar') #Search Query
urlsearchType = request.args.get('searchType') #Requested Type
urlsearchSubType = request.args.get('searchSubType') #Requested Sub-Type

urlSearchBar = removeStopWords(urlSearchBar)

urlSearchBarToken = urlSearchBar.split() #Iron Man -> [Iron,Man]
urlsearchType = urlsearchType.split() #3DPrint Game -> [3DPrint, Game]
urlsearchSubType = urlsearchSubType.split() #Character Weapon -> [Character, Weapon]
```

Then, get all products that contains search query words:

```
filter_list = [Products.productName.contains(x) for x in urlSearchBarToken]
dbQueryResult = Products.query.filter(or_(*filter_list)).all()
```

However, it does not retrieve all the products words by words. For example, if the search query is “Mannequin”, but it can also retrieve a product named “Batman” and “Banana”, because this collection of string “Mannequin” contains “Man” and “an”.

To solve this, this function will be used, it uses the re module to search for a whole word within a string:

```
def findWholeWord(x):
    return re.compile(r'\b({})\b'.format(x), flags=re.IGNORECASE).search
```

Then, we can use this function to remove any product from the list that does not fulfill the query word by word:

```
ProductsListTemp = []
for x in urlSearchBarToken:
    for y in dbQueryResult:
        if(findWholeWord(x)(y.productName) is not None and y not in ProductsListTemp):
            ProductsListTemp.append(y)
```

After checking the name of the product inside the list, the product list will be filtered by the logical expression that are previously mentioned in methodology using for-loop:

```
for x in dbQueryResult:
```

Type **or** Type:

```
elif(len(urlsearchType) != 0 and len(urlsearchSubType) == 0): #Only urlsearchType selected

    if(x.productType in urlsearchType and x not in ProductsListTemp):
        ProductsListTemp.append(x)
```

Subtype **or** Subtype:

```
elif(len(urlsearchSubType) != 0 and len(urlsearchType) == 0): #Only urlsearchSubType selected

    #Only one sub-type in product
    if(" " not in x.productSubType):
        if(x.productSubType in urlsearchSubType and x not in ProductsListTemp):
            ProductsListTemp.append(x)

    #More than one sub-type
    elif(" " in x.productSubType):
        for y in urlsearchSubType:
            if(y in x.productSubType and x not in ProductsListTemp):
                ProductsListTemp.append(x)
                break
```

Type **and** (Subtype **or** Subtype):

```
if(len(urlsearchType) != 0 and len(urlsearchSubType) != 0): #both selected

    #Start from bottom
    #Only one sub-type in product
    if(" " not in x.productSubType):
        if(x.productSubType in urlsearchSubType and x not in ProductsListTemp):
            ProductsListTemp.append(x)

    #More than one sub-type
    elif(" " in x.productSubType):
        for y in urlsearchSubType:
            if(y in x.productSubType and x not in ProductsListTemp):
                ProductsListTemp.append(x)
                break

    if(ProductsListTemp != []):
        for y in ProductsListTemp:
            if (y.productType not in urlsearchType):
                ProductsListTemp.remove(y)
```

For Type **and** (Subtype **or** Subtype), `.remove()` is to fulfill the “**and**” logic. After all the products are the requested sub-type were added to the list, then they will be removed if they are not the requested type.

The way these code work can be summarized as: creating a new list and then only take those products that contains the type or subtype and add them into the new list.

Sometimes the search bar is empty, implying user wants to search products with only types. Then, we can simply get all the products that is the requested type and sub-type

```
#If search bar is empty, but filter list is filled, than return result based on selected type and subtype
elif(len(urlsearchType) != 0 or len(urlsearchSubType) != 0):
    dbQueryResult = getProductListByTypeORSubType(urlsearchType, urlsearchSubType)
```

```
def getProductListByTypeORSubType(selectedType, selectedSubType):

    ProductsListType = []
    ProductsListSubType = []

    if(len(selectedType) != 0):
        filter_list = [Products.productType.contains(x) for x in selectedType]
        ProductsListType = Products.query.filter(or_(*filter_list)).all()

    if(len(selectedSubType) != 0):
        filter_list = [Products.productSubType.contains(x) for x in selectedSubType]
        ProductsListSubType = Products.query.filter(or_(*filter_list)).all()

    ProductsList = ProductsListType + ProductsListSubType #Combine two list together, but somehow it can remove duplicated elements.

    return ProductsList
```

After that, products inside the list need to be sorted:

```
weightForProduct = []

for x in dbQueryResult:

    currentWeight = 0.00

    for y in urlSearchBarToken:
        if(y in x.productName):
            currentWeight = currentWeight + 1.00

    weightForProduct.append(currentWeight)

print("weightForProduct: ", weightForProduct)

#[1, 3, 2, 6]
#[one, three, two, six] = list of products
#to
#[6, 3, 2, 1]
#[six, three, two, one]
#Highest weight = 1st in ranking
dbQueryResult = sortByReference(dbQueryResult, weightForProduct)
```

```
def sortByReference(lst, Ref):
    index = list(range(len(Ref)))
    index.sort(key = Ref.__getitem__)

    lst[::-1] = [lst[i] for i in index]

    return lst
```

It created the array that will be used for correspondingly storing the relevance weight for each product. For example, when I search “Banana Gun”, any product that contains either “Banana” or “Gun” will be weight at 1, and so any product that contains “Banana Gun” is weight at 2.

Lastly, the product list will be sent to PSS system to be sorted again.

```
from RS import PersonalizedSearchingResult
dbQueryResult = PersonalizedSearchingResult(urlSearchBarToken, dbQueryResult)
```

Building Customer Profile

`def createCustomerProfile():` This function is implemented in **RS.py**. Customer Profile are generated based on two situations: Have purchased something and have not been purchasing anything. And we will demonstrate the first situation first.

First, initiate an empty data frame that will be used for storing user’s rating for purchased products:

```
RateMatrix = pd.DataFrame(columns=['3DPrint', 'Game', 'PBR',
                                   'Animal', 'Character', 'Food', 'Plant', 'Weapon', 'Vehicle'])
```

Then, add rows to data frame. Each row assign ratings corresponding to type and subtype. For example, a 3D Print Animal 3D models was rated as 5. So, the row will be:

[5,0,0,5,0,0,0,0,0], and the indices will be the product ID.

```
for data in productHistories:

    rating = data.productRating

    #Ignore products that did not rated and do not take unrated products into consideration
    if(rating is not None and rating != 0):
        newRow= [0,0,0,0,0,0,0,0,0]

        CurrentProductDetail = Products.query.filter_by(productID = data.productID).first()

        if("3DPrint" in CurrentProductDetail.productType):
            newRow[0] = rating
        elif("Game" in CurrentProductDetail.productType):
            newRow[1] = rating
        elif("PBR" in CurrentProductDetail.productType):
            newRow[2] = rating

        if("Animal" in CurrentProductDetail.productSubType):
            newRow[3] = rating
        if("Character" in CurrentProductDetail.productSubType):
            newRow[4] = rating
        if("Food" in CurrentProductDetail.productSubType):
            newRow[5] = rating
        if("Plant" in CurrentProductDetail.productSubType):
            newRow[6] = rating
        if("Weapon" in CurrentProductDetail.productSubType):
            newRow[7] = rating
        if("Vehicle" in CurrentProductDetail.productSubType):
            newRow[8] = rating

        RateMatrix.loc[data.productID] = newRow
```

After that, all rows will be summed up for by each 3D model type column, and then separate into two different data frames, one is for Type, another one is for Sub-type.

```
#RateMatrix.sum() = Sum all the rate together group by the type
RateMatrixSum = np.split(RateMatrix.sum(), [3], axis=0)
RateMatrixTypeSum = RateMatrixSum[0]
RateMatrixSubTypeSum = RateMatrixSum[1]
```

By separating them together, so that they can be normalized separately.

```
Normalized_Type = preprocessing.normalize([RateMatrixTypeSum])
Normalized_SubType = preprocessing.normalize([RateMatrixSubTypeSum])
```

“`preprocessing.normalize`” is a normalize function that provided by the scikit-learn library.

After normalization, they are merged back together:

```
#Merging two table back together
df1 = pd.DataFrame(Normalized_Type, columns=['3DPrint', 'Game', 'PBR'], index=["RatedWeight"])
df2 = pd.DataFrame(Normalized_SubType, columns=['Animal', 'Character', 'Food', 'Plant', 'Weapon', 'Vehicle'], index=["RatedWeight"])

df1 = df1.swapaxes("index", "columns")
df2 = df2.swapaxes("index", "columns")

CP = pd.concat([df1, df2], axis=0, join='inner')
```

Lastly, assign the normalization value to new customer profile:

```
if(getCustomerProfile is not None):
    getCustomerProfile.Print = CP.loc["3DPrint"]["RatedWeight"]
    getCustomerProfile.Game = CP.loc["Game"]["RatedWeight"]
    getCustomerProfile.PBR = CP.loc["PBR"]["RatedWeight"]
    getCustomerProfile.Animal = CP.loc["Animal"]["RatedWeight"]
    getCustomerProfile.Character = CP.loc["Character"]["RatedWeight"]
    getCustomerProfile.Food = CP.loc["Food"]["RatedWeight"]
    getCustomerProfile.Plant = CP.loc["Plant"]["RatedWeight"]
    getCustomerProfile.Weapon = CP.loc["Weapon"]["RatedWeight"]
    getCustomerProfile.Vehicle = CP.loc["Vehicle"]["RatedWeight"]
    getCustomerProfile.TotalAnalysedProducts = RateMatrix.shape[0]

db.session.commit()
```

Now, let's demonstrate about the second situation where the user only provided interested product type and have not been purchased anything.

Because interested type and sub-type are store in user table, getting current user can be done by:

```
currentUserFilter = Users.query.filter by(id = current_user.id).first()
```

Since all interested types are stored in a single string separated by space, we can use `.split()` to convert that string back to an array:

```
localInterestedType = currentUserFilter.interestType.split()
localInterestedSubType = currentUserFilter.interestSubType.split()
```

And then, we can use for-loop to discover which kinds of type and sub-type user is interested in, assign 1 for interested type, otherwise assign 0.

<pre>if("3DPrint" in localInterestedType): getCustomerProfile.Print = 1 else: getCustomerProfile.Print = 0 if("Game" in localInterestedType): getCustomerProfile.Game = 1 else: getCustomerProfile.Game = 0 if("PBR" in localInterestedType): getCustomerProfile.PBR = 1 else: getCustomerProfile.PBR = 0</pre>	<pre>if("Animal" in localInterestedSubType): getCustomerProfile.Animal = 1 else: getCustomerProfile.Animal = 0 if("Character" in localInterestedSubType): getCustomerProfile.Character = 1 else: getCustomerProfile.Character = 0 if("Food" in localInterestedSubType): getCustomerProfile.Food = 1 else: getCustomerProfile.Food = 0</pre>
---	---

Lastly, a new customer profile will be stored to database.

However, occasionally there could be an accident where the user intentionally removes all their interested type, to avoid crashes, this code is implemented:

```
#All need to set to 0 just in case if someone deliberately decided to remove all interested type
if(getCustomerProfile is not None):

    getCustomerProfile.Print = 0
    getCustomerProfile.Game = 0
    getCustomerProfile.PBR = 0

    #####
    getCustomerProfile.Animal = 0
    getCustomerProfile.Character = 0
    getCustomerProfile.Food = 0
    getCustomerProfile.Plant = 0
    getCustomerProfile.Weapon = 0
    getCustomerProfile.Vehicle = 0
    #####
    getCustomerProfile.TotalAnalysedProducts = 0

db.session.commit()
```

When a new user that unwillingly to provide anything, this code will be executed, which is setting all value in customer profile as 0:

```
#A new user who does not provide interested product types
else:
    newCustomerProfile = CustomerProfile(
        userID = current_user.id,
        Print = 0,
        Game = 0,
        PBR = 0,
        Animal = 0,
        Character = 0,
        Food = 0,
        Plant = 0,
        Weapon = 0,
        Vehicle = 0,
        TotalAnalysedProducts = 0)

    db.session.add(newCustomerProfile)
    db.session.commit() #Confirm and update
```

Content-Based Filtering Recommendation System

def RecommendedProducts_ContentBased(PriorNumber, Numbers): This function is implemented in **RS.py**. Similar to building customer profiles, the recommendation is done differently, depending on whether user have been purchased something or not. Let's demonstrate the situation where user have bought something.

The CBF function has two parameters. As previously mentioned in methodology, CBF recommendation should be separated into two parts: prioritized recommendation and highest weight recommendation.

PriorNumber means number of products that are recommended according to highest weight type in customer profile, and the second parameter is the number of products that are recommended based on their weight which is top few highest among all products' weight.

To make prioritized recommendation that is based on highest weight type in customer profile, first we can get the customer profile, store all the weights in the data frame, then sort the weight from highest to lowest.

```
getCustomerProfile = CustomerProfile.query.filter_by(userID = current_user.id).first()

CP1 = pd.DataFrame(columns=['3DPrint', 'Game', 'PBR'], index=["RatedWeight"])
CP2 = pd.DataFrame(columns=['Animal', 'Character', 'Food', 'Plant', 'Weapon', 'Vehicle'], index=["RatedWeight"])

CP1.loc["RatedWeight"] = [getCustomerProfile.Print, getCustomerProfile.Game, getCustomerProfile.PBR]

CP2.loc["RatedWeight"] = [getCustomerProfile.Animal, getCustomerProfile.Character, getCustomerProfile.Food,
                          getCustomerProfile.Plant, getCustomerProfile.Weapon, getCustomerProfile.Vehicle]

CP1 = CP1.swapaxes("index", "columns")
CP2 = CP2.swapaxes("index", "columns")

CP1 = CP1.sort_values(by=['RatedWeight'], ascending=False)
CP2 = CP2.sort_values(by=['RatedWeight'], ascending=False)

CP = pd.concat([CP1, CP2], axis=0, join='inner')
```

Before getting products based on highest interested weight, we need to acknowledge the customer has purchased what products before, so that when we use query filter, we can avoid getting these products to recommendation list:

```
PurchasedProductsList = PurchaseHistory.query.filter_by(userID = current_user.id).all()
PurchasedProductsIDList = [] #Take the IDs of purchased products

for x in PurchasedProductsList:
    PurchasedProductsIDList.append(x.productID)
```

After that, we can start to use query filter to get prioritized products then add them into recommendation list "RecommendProductList" :

```
#Get the product whose type and subtype has the highest weight, and have not been purchased
#Type -> [0]
#Type -> [1]
#Type -> [2]
#Sub-Type -> [3]
#limit(PriorNumber) = only return number of (PriorNumber)
PrioritizedRecommendations = Products.query.filter(
    and_(Products.productType == CP.iloc[0].name), (Products.productSubType == CP.iloc[3].name),
    (Products.productID.notin_(PurchasedProductsIDList))
).limit(PriorNumber).all()

RecommendProductsList.extend(PrioritizedRecommendations)
```

The coding comments in the picture has visually explained the functionality behind the code. After sorting the weight in the data frame, the highest weight for product type is at the index [0] because there is only 3 product type, and the highest weight for product sub-type is at [3]. And, to avoid the query retrieve purchased products, `notin_` is utilized to exclude purchased products. `.limit()` is to limit the number of return products.

After prioritized recommendation list is created, we can start to make highest weight recommendation. We need to first get all the unpurchased products that is based on type and subtype of product the user purchased, it is unnecessary to take all products from database to calculate the weight:

```
#To find what type and subtype of product user purchased:
purchasedType = np.array([])
purchasedSubType = np.array([])
if(CP.loc["3DPrint"]["RatedWeight"] > 0):
    purchasedType = np.append(purchasedType, "3DPrint")
if(CP.loc["Game"]["RatedWeight"] > 0):
    purchasedType = np.append(purchasedType, "Game")
if(CP.loc["PBR"]["RatedWeight"] > 0):
    purchasedType = np.append(purchasedType, "PBR")

if(CP.loc["Animal"]["RatedWeight"] > 0):
    purchasedSubType = np.append(purchasedSubType, "Animal")
if(CP.loc["Character"]["RatedWeight"] > 0):
    purchasedSubType = np.append(purchasedSubType, "Character")
if(CP.loc["Food"]["RatedWeight"] > 0):
    purchasedSubType = np.append(purchasedSubType, "Food")
if(CP.loc["Plant"]["RatedWeight"] > 0):
    purchasedSubType = np.append(purchasedSubType, "Plant")
if(CP.loc["Weapon"]["RatedWeight"] > 0):
    purchasedSubType = np.append(purchasedSubType, "Weapon")
if(CP.loc["Vehicle"]["RatedWeight"] > 0):
    purchasedSubType = np.append(purchasedSubType, "Vehicle")

ProductsList = getProductListByTypeORSubType(purchasedType, purchasedSubType)
```

After that, we create a data frame that store all the weights of the unpurchased products and their corresponding id:

```
ProductWeights = pd.DataFrame(columns=["ProductID", "ProductName", "Weight", "Type", "Sub-Type"])
```

Then, using for loop to calculate the weights:

```
newRow= [0,0,0, "", ""]

#Dont include those are already purchased and inside the prior recommendation list
if(PLists.productID not in PurchasedProductsIDList and PLists not in PrioritizedRecommendations):
    newRow[0] = PLists.productID
    newRow[1] = PLists.productName

    if("3DPrint" in PLists.productType):
        newRow[2] = CP.loc["3DPrint"]["RatedWeight"]
        newRow[3] = "3DPrint"
    elif("Game" in PLists.productType):
        newRow[2] = CP.loc["Game"]["RatedWeight"]
        newRow[3] = "Game"
    elif("PBR" in PLists.productType):
        newRow[2] = CP.loc["PBR"]["RatedWeight"]
        newRow[3] = "PBR"

    if("Animal" in PLists.productSubType):
        newRow[2] += CP.loc["Animal"]["RatedWeight"]

    if("Character" in PLists.productSubType):
        newRow[2] += CP.loc["Character"]["RatedWeight"]

    if("Food" in PLists.productSubType):
        newRow[2] += CP.loc["Food"]["RatedWeight"]

    if("Plant" in PLists.productSubType):
        newRow[2] += CP.loc["Plant"]["RatedWeight"]

    if("Weapon" in PLists.productSubType):
        newRow[2] += CP.loc["Weapon"]["RatedWeight"]

    if("Vehicle" in PLists.productSubType):
        newRow[2] += CP.loc["Vehicle"]["RatedWeight"]

    newRow[4] += PLists.productSubType

    ProductWeights.loc[PLists.productID] = newRow
```

After calculating the weights, we sort the rows inside the data frame by the weight, and then get the products items back and put them into the recommendation list

“RecommendProductList”:

```
ProductWeights = ProductWeights.sort_values(by=['Weight'], ascending=False).reset_index(drop=True)

TopProducts = ProductWeights.head(Numbers)
for x in range(Numbers):
    TheOtherRecommendedProduct = Products.query.filter(Products.productID == TopProducts.loc[x].ProductID).first()
    RecommendProductsList.append(TheOtherRecommendedProduct)
```

And that’s how CBF recommendation is done for users who have purchased items.

Now, let’s demonstrate how CBF recommendation is done for those who have only provided their interested item type:

```
#To find what type and subtype of product user purchased:
purchasedType = np.array([])
purchasedSubType = np.array([])
if(getCustomerProfile.Print > 0):
    purchasedType = np.append(purchasedType, "3DPrint")
if(getCustomerProfile.Game > 0):
    purchasedType = np.append(purchasedType, "Game")
if(getCustomerProfile.PBR > 0):
    purchasedType = np.append(purchasedType, "PBR")

if(getCustomerProfile.Animal > 0):
    purchasedSubType = np.append(purchasedSubType, "Animal")
if(getCustomerProfile.Character > 0):
    purchasedSubType = np.append(purchasedSubType, "Character")
if(getCustomerProfile.Food > 0):
    purchasedSubType = np.append(purchasedSubType, "Food")
if(getCustomerProfile.Plant > 0):
    purchasedSubType = np.append(purchasedSubType, "Plant")
if(getCustomerProfile.Weapon > 0):
    purchasedSubType = np.append(purchasedSubType, "Weapon")
if(getCustomerProfile.Vehicle > 0):
    purchasedSubType = np.append(purchasedSubType, "Vehicle")

RecommendProductsList = getProductListByTypeORSubType([purchasedType, purchasedSubType])
```

The way it works is much simpler to previous method, first we discovered what kind of type the user is interested in, and then we get all the products based on those interested type.

After that, we randomly pick few items from those product list to recommend to user:

```
#Randomly pick products from list
K_ = PriorNumber + Numbers
if(len(RecommendProductsList) >= K_): #Sometimes k value could be larger than the # of products inside RecommendProductsList
    RecommendProductsList = random.sample(RecommendProductsList, k = K_ )
else:
    RecommendProductsList = random.sample(RecommendProductsList, k = len(RecommendProductsList) )

return RecommendProductsList
```

Finding Similar Users

def FindSimilarUser_CF(): This function is implemented in **RS.py**. To recommended product with CFRS. We need to find similar user first.

First, we get other customer profiles that has analyze at least one purchase product:

```
AllCustomerProfile = CustomerProfile.query.filter( and_(CustomerProfile.TotalAnalysedProducts > 0), (CustomerProfile.userID != current_user.id) ).all()
```

However, if the system only contains two users, the system can immediately get that user without wasting computational power.

```
#If there is only one other user in database, we can save time from KNN
if(len(AllCustomerProfile) == 1):

    #Find the other user
    SimilarUsers = np.array([Users.query.filter_by( id = x.userID ).first()])

    return SimilarUsers
```

And we take the customer profile of targeted user:

```
TargetedCustomerProfile = CustomerProfile.query.filter_by(userID = current_user.id).first()
```

After that, we store the weight values into arrays.

```

CustomerProfiles_Array = np.empty((0, 9)) #Array for weights

TargetProfile_Array = np.array([TargetedCustomerProfile.Print,
                                TargetedCustomerProfile.Game,
                                TargetedCustomerProfile.PBR,
                                TargetedCustomerProfile.Animal,
                                TargetedCustomerProfile.Character,
                                TargetedCustomerProfile.Food,
                                TargetedCustomerProfile.Plant,
                                TargetedCustomerProfile.Weapon,
                                TargetedCustomerProfile.Vehicle])

for profile in AllCustomerProfile:
    temp = np.array([profile.Print, profile.Game, profile.PBR, profile.Animal, profile.Character, profile.Food, profile.Plant, profile.Weapon, profile.Vehicle])
    CustomerProfiles_Array = np.append(CustomerProfiles_Array, [temp], axis = 0)

```

Then, kNN model will be utilized to find similar users, by using 'auto' algorithm, the algorithm will automatically choose the suitable algorithm for the inputted data

```

knnModel = NearestNeighbors(n_neighbors= K, algorithm='auto').fit(CustomerProfiles_Array)
distance, sortedSimilarUsersIndex = knnModel.kneighbors(TargetProfile_Array.reshape(1, -1))

```

kNN will then return the indices of similar users in descending order, then we can find the similar user:

```

#Pick user by customer profile
SimilarUsers = np.array([])
for x in SimilarCustomerProfiles_Array:
    SimilarUser = Users.query.filter_by( id = x.userID ).first()
    SimilarUsers = np.append(SimilarUsers, [SimilarUser], axis=0)

```

Collaborative Filtering Recommendation System

`def RecommendProduct_CollaborativeFiltering(RecNum, lowestRating):` This function is

implemented in **RS.py**. RecNum is number of products to be recommended, lowestRating will make CF to pick products that is not lower than the rating value made by similar user.

After finding similar user with: `SimilarUsers = FindSimilarUser_CF()`, we can start to find their products.

To find all products that purchased by similar user, we need to use for loop for their purchase history:

```
i = 0
Nestbreak = False
RecommendUserProduct = []
addedProducts = [] # keep track of added products to avoid duplication

# Recommend products that are rated 5 to 4, from similar users
for x in SimilarUsers:
    SelectedPurchasedProducts_Query = PurchaseHistory.query.filter(and_(PurchaseHistory.userID == x.id), (PurchaseHistory.productRating >= lowestRating) ).limit(RecNum).all()

    Temp = np.array([])
    Temp = np.append(Temp, [x], axis = 0)
    for queryProductHistory in SelectedPurchasedProducts_Query:

        # To check whether the targeted user has purchased the recommended product or not
        if( PurchaseHistory.query.filter( and_(PurchaseHistory.productID == queryProductHistory.productID) , (PurchaseHistory.userID == current_user.id) ).first() is None):
            product = Products.query.filter_by(productID = queryProductHistory.productID).first()

            # Check if the product is already in the added_products list
            if(product.productID not in addedProducts):
                Temp = np.append(Temp, [queryProductHistory], axis = 0)
                Temp = np.append(Temp, [product], axis = 0)

                addedProducts.append(product.productID) # add the product ID to addedProducts
                i += 1

            if i == RecNum:
                Nestbreak = True
                break

    RecommendUserProduct.append(Temp.tolist())

if Nestbreak:
    break

# Output example: [[<Users 3>], [<Users 2>, <PurchaseHistory 3>, <Products 12>, <PurchaseHistory 4>, <Products 9>]]
return RecommendUserProduct
```

Personalized Search System

def PersonalizedSearchingResult(SearchQueryToken, OrginialSearchProductResults): This function is implemented in **PSS.py**. It will be executed after the basic search system has sort the search results. The reason the basic search system also sort the result is that it will be a lot easier for PSS to further divide the search results into three groups:

```
#To break down OrginialSearchProductResults into two section: All fulfilled + Query fulfilled + Partially fulfilled
i = 0
NestBreak = False
AllFulfilledResult = 0
for OProducts in OrginialSearchProductResults:

    for x in range(len(SearchQueryToken)):
        if(SearchQueryToken[x] not in OProducts.productName):
            NestBreak = True
            break

        elif(x == (len(SearchQueryToken)-1) and len(SearchQueryToken) == len(OProducts.productName.split()) ):
            AllFulfilledResult = OProducts

    if(NestBreak == True):
        break
    else:
        i=i+1
```

Basically, this part is to get the i value that is a boundary line between Query fulfilled and Partially fulfilled:

```
QueryFulfilledResult = OrginialSearchProductResults[:i]
PartiallyFulfilledResult = OrginialSearchProductResults[i:]
```

After that, three sections were created, we can start to personalize the search result in Query fulfilled and Partially fulfilled section separately. This will create a list of arrays that contains weight to corresponding products:

```
QueryFWeights = SearchedProductWeightList(QueryFulfilledResult)
PartWeights = SearchedProductWeightList(PartiallyFulfilledResult)

def SearchedProductWeightList(ProductList):

    getCustomerProfile = CustomerProfile.query.filter_by(userID = current_user.id).first()
    weightForProduct = np.array([])

    #Give weight to each result
    for x in ProductList:

        if("3DPrint" in x.productType):
            weight = getCustomerProfile.Print
        elif("Game" in x.productType):
            weight = getCustomerProfile.Game
        elif("PBR" in x.productType):
            weight = getCustomerProfile.PBR
        elif("" in x.productType):
            weight = 0

        if("Animal" in x.productSubType):
            weight += getCustomerProfile.Animal

        if("Character" in x.productSubType):
            weight += getCustomerProfile.Character

        if("Food" in x.productSubType):
            weight += getCustomerProfile.Food

        if("Plant" in x.productSubType):
            weight += getCustomerProfile.Plant

        if("Weapon" in x.productSubType):
            weight += getCustomerProfile.Weapon

        if("Vehicle" in x.productSubType):
            weight += getCustomerProfile.Vehicle

        if("" in x.productSubType):
            weight += 0

        weightForProduct = np.append(weightForProduct, weight)

    return weightForProduct
```

It uses customer profile to calculate sum the weights for each product.

After sorting Query fulfilled and Partially fulfilled section separately. We then merge all three sections together to finalize the personalized search result.

```
sortedQueryF = sortByReference(QueryFulfilledResult, QueryFWeights)
sortedPart = sortByReference(PartiallyFulfilledResult, PartWeights)

Finallist = np.concatenate((sortedQueryF, sortedPart), axis=0)

if(AllFulfilledResult != 0 and len(Finallist) != 0):
    Finallist = np.insert(Finallist, 0, AllFulfilledResult)
elif(AllFulfilledResult != 0 and len(Finallist) == 0):
    Finallist = np.array([AllFulfilledResult])
```


Evaluation

Result

Product Recommendation

`RecommendedProducts_ContentBased(5, 5)` `RecommendProduct_CollaborativeFiltering(10, 4)`

This is the parameter for CF and CBF, each of them will recommend 10 items.

Let's create a new user named "U9" to the store, his user id will be 9, and assume he is interested in 3D Print Animal Character 3D Models:

Add User

☒ Male ☐ Female

Interest

Type:
☒ 3D Print
 ☐ Game
 ☐ PBR

Sub-Type Model:
☒ Animal
 ☒ Character
 ☐ Food
 ☐ Plant
 ☐ Weapon
 ☐ Vehicle

And this will be his current profile

3DPrint	Game	PBR	Animal	Character	Food	Plant	Weapon	Vehicle	Rated Products
1.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0

Without buying items, his CBF recommendations list is:

You might be interested in:			
ID	Name	Type	Sub-Type
70	Cat	PBR	Animal
87	Duck	PBR	Animal
148	Snake Plant	3DPrint	Plant
113	Cake	3DPrint	Food
78	Centipede	Game	Animal
106	Mickey Mouse	3DPrint	Animal Character
91	Dormouse	3DPrint	Animal
88	Dragonfly	Game	Animal
89	Dodo	3DPrint	Animal
61	Anakin Skywalker Bust	3DPrint	Character

All the recommended product is all relevant U9's interested types: 3D Print, Animal, Character.

And his CF recommendations list is

Similar user have also bought those:					
Similar User ID	User Rating	Product ID	Product Name	Type	Sub-Type
7	5	85	Dog	3DPrint	Animal
7	5	89	Dodo	3DPrint	Animal
7	5	102	Pikachu	3DPrint	Character
7	5	82	Golden Cow	Game	Animal Character
5	4	19	Lion	3DPrint	Animal
5	5	27	Fox	3DPrint	Animal
5	4	20	Bee	PBR	Animal
5	5	3	Toy Story - Woody	Game	Character
2	5	28	DoomSlayer	Game	Character
2	5	12	Pistol	3DPrint	Weapon

It has indicated that the id of similar user is: 7,5,2. Let's see their profile in database:


userID	Print	Game	PBR	Animal	Character	Food	Plant	Weapon	Vehicle
2	0.744208	0.620174	0.248069	0.0982946	0.933799	0	0	0.344031	0
5	0.698536	0.46569	0.543305	0.959365	0.282166	0	0	0	0
7	0.945905	0.278207	0.166924	0.754732	0.603786	0.150946	0.150946	0.100631	0.100631
9	1	0	0	1	1	0	0	0	0

These users have common interests, they prone to have higher weight in Print, Animal and Character. In kNN perspective, their distance to U9 is:

```
TargetUserID: 9
KNN Result:
Distance: 0.6254      User ID: 7
Distance: 1.0582     User ID: 5
Distance: 1.2031     User ID: 2
```

Because User 7 is the most similar to U9, so, his purchased products is being recommended at the top of the list.

Now, assume U9 has purchased some products. However, his interests were changed a bit, so he bought and highly rated Game Character 3D Models more:

Purchased History				
Show 				
ID	Rating	Type	Sub-Type	Name
107	5	Game	Character	Luke Skywalker
11	4	Game	Character	Mr Bean
101	5	3D Print	Character	Mario
8	5	3D Print	Character	War Machine
93	5	Game	Animal Character	Crocodile
82	5	Game	Animal Character	Golden Cow
22	2	3D Print	Character	Devil
15	5	Game	Animal Character	Monke

And his new profile, the highest weight of type and sub-type is now **Game Character**.

3DPrint	Game	PBR	Animal	Character	Food	Plant	Weapon	Vehicle	Rated Products
0.447214	0.894427	0.0	0.384615	0.923077	0.0	0.0	0.0	0.0	8

After that, the CB recommend products are mostly Game Character models, and Game animal model occasionally.

You might be interested in:			
ID	Name	Type	Sub-Type
2	Iron Man	Game	Character
3	Toy Story - Woody	Game	Character
21	Ghost	Game	Character
28	DoomSlayer	Game	Character
31	Joker	Game	Character
92	Duck-billed Platypus	Game	Animal Character
95	Chimpanzee	Game	Animal Character
207	The Epona (Legend of Zelda)	Game	Character Vehicle
43	Master Chief	Game	Character
66	Evangelion Shinji	Game	Character

```

(Content-Based) Prioritized Recommendation:
2 : Iron Man
3 : Toy Story - Woody
21 : Ghost
28 : DoomSlayer
31 : Joker
(Content-Based) Product Weight List:
ProductID      ProductName      Weight      Type      Sub-Type
0      92      Duck-billed Platypus  2.202119  Game      Animal Character
1      95      Chimpanzee      2.202119  Game      Animal Character
2      207      The Epona (Legend of Zelda)  1.817504  Game      Character Vehicle
3      43      Master Chief      1.817504  Game      Character
4      66      Evangelion Shinji  1.817504  Game      Character
..      ...      ...      ...      ...
158     46      Stag Beetle      0.384615  PBR      Animal
159     44      Blue Butterfly    0.384615  PBR      Animal
160     17      Dolphin          0.384615  PBR      Animal

```

And the new similar users are:

```

KNN Result:
Distance: 0.5230      User ID: 10
Distance: 0.5694      User ID: 11
Distance: 0.6522      User ID: 2

```

Their Customer Profile, they prone to have higher weight in Game and Character.

userID	Print	Game	PBR	Animal	Character	Food	Plant	Weapon	Vehicle
2	0.744208	0.620174	0.248069	0.0982946	0.933799	0	0	0.344031	0
10	0.179425	0.897123	0.403705	0.196116	0.980581	0	0	0	0
11	0	0.966235	0.257663	0.278187	0.938882	0.10432	0	0	0.173867

And the new CF recommendation list:

Similar user have also bought those:					
Similar User ID	User Rating	Product ID	Product Name	Type	Sub-Type
10	5	2	Iron Man	Game	Character
10	5	31	Joker	Game	Character
10	4	103	Captain America	Game	Character
10	5	33	Flash	PBR	Character
10	4	9	Darth Vader	PBR	Character
11	5	21	Ghost	Game	Character
11	5	28	DoomSlayer	Game	Character
11	5	97	Harry Potter	Game	Character
11	5	216	F-150	PBR	Vehicle
2	5	12	Pistol	3DPrint	Weapon

Personalized Searching

Let's continue to use U9's customer profile to search products. Assume he wants to search "Iron Man" :

Search Result			
ID	Name	Type	Sub-Type
2	Iron Man	Game	Character
58	Iron Man Mark 42	Game	Character
57	Iron Man Gauntlet	3DPrint	Character Weapon
60	Iron Man Mark 7 Helmet	3DPrint	Character
219	Iron Man Mark 3	PBR	Character
105	Spider-Man	Game	Character
23	Spider Man 2099	PBR	Character
59	Iron Ingot	Game	

```

All fulfilled section: <Products 2>
Query fulfilled section: [<Products 219>, <Products 60>, <Products 58>, <Products 57>]
Partially fulfilled section: [<Products 105>, <Products 23>, <Products 59>]
QWeight [0.923077 1.370291 1.817504 1.370291]
PWeight [1.817504 0.923077 0.894427]

```

User's 9 profile:

3DPrint	Game	PBR	Animal	Character	Food	Plant	Weapon	Vehicle	Rated Products
0.447214	0.894427	0.0	0.384615	0.923077	0.0	0.0	0.0	0.0	8

Each product in the search result list has their own weight calculated by summing all the corresponding user's interested type:

Query Fulfilled Product	Weight	Rank	Partially Fulfilled Product	Weight	Rank
219	0.923077	4	105	1.817504	1
60	1.370291	3	23	0.923077	2
58	1.817504	1	59	0.894427	3
57	1.370291	2			

Table 7: Calculation of weights for searched products

Therefore, the higher the product in the search list, the more relevant to the user's needs.

Now, let switch to another user U5, and search "Iron Man" again:

Search Result			
ID	Name	Type	Sub-Type
2	Iron Man	Game	Character
57	Iron Man Gauntlet	3DPrint	Character Weapon
60	Iron Man Mark 7 Helmet	3DPrint	Character
219	Iron Man Mark 3	PBR	Character
58	Iron Man Mark 42	Game	Character
23	Spider Man 2099	PBR	Character
105	Spider-Man	Game	Character
59	Iron Ingot	Game	

User's 5 profile:

3DPrint	Game	PBR	Animal	Character	Food	Plant	Weapon	Vehicle	Rated Products
0.698536	0.46569	0.543305	0.959365	0.282166	0.0	0.0	0.0	0.0	6

```

All fulfilled section: <Products 2>
Query fulfilled section: [<Products 219>, <Products 60>, <Products 58>, <Products 57>]
Partially fulfilled section: [<Products 105>, <Products 23>, <Products 59>]

QWeight [0.825471 0.980702 0.747856 0.980702]
PWeight [0.747856 0.825471 0.46569 ]

```

Query Fulfilled Product	Weight	Rank	Partially Fulfilled Product	Weight	Rank
219	0.825471	3	105	0.747856	2
60	0.980702	2	23	0.825471	1
58	0.747856	4	59	0.46569	3
57	0.980702	1			

Table 8: Calculation of weights for searched products

Compared to U9's search results, the search list is mostly different, that is because U5's interested product type is different than U9.

Recommendation System Drawbacks

There is a drawback in this RS with CF approach, imbalance population. This is similar to **Popularity Based** problem mentioned in literature review, and it is noticeable when the online store is recently published. For example, there are 30 customers in the store, but 22 of them are mostly interested in 3D Print models, and the last of them interested in either Game or PBR models, the problem is that those few users could also being recommended 3D print models frequently. Because in kNN, those "unsimilar" users could happen to have

the close distance to targeted user, which will make those unsimilar user become the most similar users to targeted user. But this drawback will become less noticeable when there is more user in the store, except the diversity of recommended products for minor group might be as good as major group.

Another drawback is from the perspective of user, CB recommend product obviously. This will more likely become problematic when the store only has very few items. When a user is interested in certain type of items, they could search the items by themselves, which made the RS lack of novelty and helpfulness. The problem can be alleviated when the store contains more 3D models.

Discussion

Large Dataset Expansion Solution

The scale of the dataset in this online store will be very large eventually. The current RS system will then demand a larger computational power. If the database is stored in a cloud environment where computational resources are charged based on usage, the price of hosting will be higher. To solve this problem, we need to address two recommendation lists separately.

For CBF, the store can build a SVM model to build a customer profile with the user's ratings from "purchased_history" and "users" table, so that not only will SVM be able to predict user's interests, but also PSS can utilize the customer profile to sort the searching results. It is possible to use other machine learning models, however, it must be able to be cooperate with PSS.

For CF, the store should export the kNN model to reduce computational power, because kNN is known for demanding large computational power for large dataset. Every time when we make prediction, the kNN model will need to be trained from scratch, it creates index that has all the position of all data point in memory to find the k-nearest neighbors to new input data point. [] Therefore, by exporting the model, it will not need to repeatedly train the model from scratch to find similar users, because the model's parameters and metadata are saved, including the index that was created during training, the exported model will use the saved index to find k-nearest neighbors for new input data instead of loading all the original input data points in memory again.[] Therefore, exporting the kNN model once the scale of the data is large will saving computational power and memory usage.

Conclusion

To conclude, we have built a RS and PSS for the online store that is still in-development without the real-life dataset.

In literature review, three common publicly available RS system approaches were mentioned: Popularity, Content-Based Filtering (CBF) , and Collaborative Filtering (CF). We took these approaches as reference to begin the development of RS and PSS. The functionality of CBF can be briefly summarized as it explores users interested on certain type of items to recommend products, and as for CF, it finds similar users, and then recommend their highly rated purchased products to targeted user. And for popularity, it recommends products solely based on their most favorable aspect, such as highest sales, rather than based on other important factors.

After the investigation on their functionalities in practice in methodology, we concluded that our RS for the online store is more suitable to adopt CBF and CF as they will make RS has more innovation potential to evolve in the future, when it has enough real-life dataset.

Customer profile is the core of both RS and PSS, it records and calculating the weight of the types of products the user is interested in by taking their ratings on purchased products as inputs. By utilizing it, RS can recommend products and personalizing search result Based on the weight provided by customer profile.

And, to find similar user effectively with the very small dataset in practice, kNN machine learning model was used as it can take a very small dataset to find similar users by utilizing and comparing customer profiles. Although the scale of the store will expand in the future, which will increase computational power. It can be solved by exporting and redeploying the model because it will no longer needs to be repeatedly retrained from scratch again to make prediction.

The drawbacks of RSS are imbalanced interested in population in CF and lack of existing purpose of CBF, they could be very likely to happen when the store is recently published and only have very few real-life data. However, both problems will become less noticeable and will be alleviated once the size of the store is increased, which means there will be more real-life dataset for building a better RSS with machine learning model.

Reference

- [1] B. MORISET, "e-Business and e-Commerce," 12 4 2018. [Online]. Available: <https://shs.hal.science/halshs-01764594/#>.
- [2] M. L. Solvang and S. Sand, "Video Recommendation Systems - Finding a Suitable Recommendation Approach for an Application Without Sufficient Data," 8 2017. [Online]. Available: <https://www.duo.uio.no/bitstream/handle/10852/59239/1/MariusLorstadSolvangSteffenSand.pdf>.
- [3] P. Pu, L. Chen and R. Hu, "A user-centric evaluation framework for recommender systems," 2011.
- [4] L. Liu, "e-Commerce Personalized Recommendation Based on Machine Learning Technology," IOS Press, Netherlands, 2022.
- [5] L. E. M. FERNÁNDEZ, "Recommendation System for Netflix," 29 1 2018. [Online]. Available: <https://www.cs.vu.nl/~sbhulai/papers/paper-fernandez.pdf>.
- [6] "Content-based Filtering Advantages & Disadvantages," Google, [Online]. Available: <https://developers.google.com/machine-learning/recommendation/content-based/summary#:~:text=The%20model%20can%20only%20make,on%20the%20user's%20existing%20interests>.
- [7] "What is the k-nearest neighbors algorithm?," IBM, [Online]. Available: <https://www.ibm.com/topics/knn#:~:text=The%20k%2Dnearest%20neighbors%20algorithm%2C%20also%20known%20as%20KNN%20or,of%20an%20individual%20data%20point..>
- [8] M. Hahsler, "recommenderlab: An R Framework for Developing," [Online]. Available: <https://cran.r-project.org/web/packages/recommenderlab/vignettes/recommenderlab.pdf>.