Skincare Write-Up

Sofia Ward / Bowie Chuang / Christina Pham / Carter Kulm

2025-03-11

Curated Sephora Skincare Generator 3000000

Content Based Filtering Recommendation System

Introduction

With the growing number of skincare products available, choosing the right one for an individual's specific needs can be overwhelming. To help users navigate this vast selection, we developed a content-based skincare recommendation system that suggests products based on their attributes and similarity to other items.

We wanted to build something that would save users money and time from trying product after product, since everyone's skin is different! Advertising and reviews are misleading since company goals don't align with the user's best interest: finding the right skincare! Our model not only selects products based on similarity, but also incorporates ingredient lists, skin types, and targeted skin concerns. This filtering gives users 10 individually catered products to their specific needs. For user accessibility we chose Sephora's Skincare selection.

Web-scraped Data Description

To build our skincare recommendation system, we collected product data from the Sephora website using web scraping techniques. After checking for permissions, we used the robots.txt file to acquire an html of product links, utilizing Selenium's Webdriver and Beautiful Soup. Filtering cosmetics and hair products left us with a categorized link data frame to begin web-scraping!

We encountered several obstacles but through trial and error extracted our information; pop-up ads, scroll-down functionality, custom user-agent strings to avoid bot detection. Scraping itself took copious amounts of time, and we found that smaller segments extracted less N/A's. Altogether we scraped **1,800 links** which filtered out to **500 observations and 12 columns.** The final scraping loop is provided in the appendix of this report.

```
## [1] 532  12
```

```
## Rows: 532
## Columns: 12
## $ `Brand Name`          <chr> "Dr. Barbara Sturm", "Herbivore", "Dr. Dennis Gr~
## $ `Product Name`        <chr> "Cleanser", "Aquarius BHA + Blue Tansy Clarifyin~
## $ `Product Category`    <chr> "Cleanser", "Cleanser", "Cleanser", "Cleanser", ~
## $ `Product Price`       <dbl> 108.0, 26.0, 39.0, 52.0, 18.0, 7.0, 40.0, 24.0, ~
## $ `Product Rating`      <dbl> 4.1, 4.6, 4.3, 4.4, 4.6, 4.3, 4.5, 4.3, 4.6, 4.6~
## $ `Product Size`        <chr> "5 oz/ 150 mL", "3.38 oz / 100 mL", "7.5 oz/ 225~
## $ `Product Reviews`     <dbl> 74, 269, 486, 58, 498, 448, 2948, 5380, 45, 498,~
## $ `Product Description` <chr> "A gentle foaming cleanser that removes makeup, ~
## $ `Product Ingredients` <chr> "-Mild Tensides: Provide thorough, yet gentle cl~
## $ `Skin Type`           <chr> "Normal, Dry, Combination, and Oily", "Normal, C~
## $ `Skin Concerns`       <chr> "Dryness, Fine Lines and Wrinkles, and Acne and ~
## $ URL                   <chr> "https://www.sephora.com/ca/en/product/dr-barbar~
```

The data set contains key attributes essential for content-based filtering, including:

- **Product Name**: The name of the skincare product.
- **Brand**: The company or brand that manufactures the product.
- **Category**: The type of product: exfoliates, cleansers, toners, serums, moisturizer, masks).
- **Price**: The cost of the product in USD.
- **Reviews**: The number of user reviews for the product.
- **Size**: The quantity of the product (e.g., 100ml, 1.7oz).
- **Ingredients**: A list of active and inactive ingredients.
- **Description**: A textual summary of the product, often provided by the brand or Sephora.
- **Skin Concern**: The specific skin issues the product addresses (e.g., acne, dryness, hyper-pigmentation).
- **Skin Type**: The recommended skin types for the product (e.g., oily, dry, combination).

Methodology

Data Pre-processing

After collecting the raw data, we performed several pre-processing steps:

1. **Cleaning the Data**: Removed duplicate entries and handled missing values through imputation or deletion.

```
## [1] 0
```

2. **Feature Engineering**: Formatted price, rating, and size for consistency.

```
## # A tibble: 6 x 12
##   `Brand Name`              `Product Name`    `Product Category` `Product Price`
##   <chr>                     <chr>             <chr>                        <dbl>
## 1 Dr. Barbara Sturm         Cleanser          Cleanser                       108
## 2 Herbivore                 Aquarius BHA + B~ Cleanser                        26
## 3 Dr. Dennis Gross Skincare Alpha Beta® AHA/~ Cleanser                        39
## 4 Hourglass                 Equilibrium Reb~ Cleanser                       52
```

```
## 5 Benefit Cosmetics         Mini The POREfes~ Cleanser                          18
## 6 The INKEY List            Mini Fulvic Acid~ Cleanser                           7
## # i 8 more variables: `Product Rating` <dbl>, `Product Size` <chr>,
## #   `Product Reviews` <dbl>, `Product Description` <chr>,
## #   `Product Ingredients` <chr>, `Skin Type` <chr>, `Skin Concerns` <chr>,
## #   URL <chr>
```

Recommendation System: Content-Based Filtering

Our system uses **content-based filtering**, a recommendation approach that suggests products based on their attributes rather than user interactions. We opted for this approach to respect the privacy of Sephora's customers and for user accessibility.

What is Content-Based Filtering?

Content-based filtering analyzes the characteristics of items to recommend similar products. Instead of relying on user behavior (as in collaborative filtering), it compares the features of a given product to those of other products in the data set.

In our case, we represent each skincare product as a **feature vector**, incorporating product descriptions, ingredients, category, skin concerns, and other relevant attributes. The similarity between products is calculated using **cosine similarity**, which measures the angle between two vectors in a multidimensional space. A higher cosine similarity score indicates that two products are more alike.

Mathematically, cosine similarity between two product vectors **A** and **B** is given by:

$$S_C(A, B) = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

Where:

- $A \cdot B$ is the dot product of the two vectors,
- $\|A\|$ and $\|B\|$ are the magnitudes of the vectors.

Using this approach, when a user selects a product, the system finds other products with the highest cosine similarity, ensuring recommendations are tailored to the product's attributes!

Implementation

We implemented the recommendation system in **Python**, using `scikit-learn` for text vectorization and similarity computations. The key steps include:

1. **Text Vectorization**: We converted textual data (e.g., ingredients, descriptions) into vectors using **TF-IDF (Term Frequency-Inverse Document Frequency)**.

```
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
import openpyxl
import string

df = pd.read_excel("/Users/fifi/Desktop/Skin_Care_Recommendation/data/filtered_sephora.xlsx")
df
```

```
##                     Brand Name  ...                                        URL
## 0            Dr. Barbara Sturm  ...  https://www.sephora.com/ca/en/product/dr-barba...
## 1                     Herbivore  ...  https://www.sephora.com/ca/en/product/herbivor...
## 2    Dr. Dennis Gross Skincare  ...  https://www.sephora.com/ca/en/product/dr-denni...
```

```
## 3                   Hourglass  ...  https://www.sephora.com/ca/en/product/hourglas...
## 4           Benefit Cosmetics  ...  https://www.sephora.com/ca/en/product/benefit-...
## ..                        ...  ...                                                ...
## 527                   La Mer  ...  https://www.sephora.com/ca/en/product/la-mer-t...
## 528              The Ordinary  ...  https://www.sephora.com/ca/en/product/the-ordi...
## 529         Dr. Barbara Sturm  ...  https://www.sephora.com/ca/en/product/dr-barba...
## 530         iNNBEAUTY PROJECT  ...  https://www.sephora.com/ca/en/product/innbeaut...
## 531                    Murad  ...  https://www.sephora.com/ca/en/product/murad-mi...
##
## [532 rows x 12 columns]
```

```
df_recommend2 = df[["Brand Name", "Product Name", "Product Category", "Product Description",
                    "Product Ingredients", "Skin Type", "Skin Concerns"]]
df_recommend2_cols = ["Brand Name", "Product Name", "Product Category", "Product Description",
                    "Product Ingredients", "Skin Type", "Skin Concerns"]
df_recommend2.columns = df_recommend2_cols
# Select columns
# concatenate all the strings and then fit it into TfidfVectorizer

df_recommend2['Product Description'] = df_recommend2['Product Description'].fillna(value = "No Descripti
```

```
## <string>:4: SettingWithCopyWarning:
## A value is trying to be set on a copy of a slice from a DataFrame.
## Try using .loc[row_indexer,col_indexer] = value instead
##
## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexi
```

```
df_recommend2
```

```
##                    Brand Name  ...                                  Skin Concerns
## 0           Dr. Barbara Sturm  ...  Dryness, Fine Lines and Wrinkles, and Acne and...
## 1                    Herbivore  ...            Pores, Acne and Blemishes, and Oiliness
## 2    Dr. Dennis Gross Skincare  ...                Pores, Dullness, and Uneven Texture
## 3                    Hourglass  ...                              Dryness and Dullness
## 4           Benefit Cosmetics  ...                                             Pores
## ..                        ...  ...                                               ...
## 527                   La Mer  ...                                  Pores, Oiliness
## 528              The Ordinary  ...      Pores, Uneven Texture, and Acne and Blemishes
## 529         Dr. Barbara Sturm  ...      Fine Lines and Wrinkles, Pores, and Blemishes
## 530         iNNBEAUTY PROJECT  ...   Fine Lines/Wrinkles, Dryness, and Uneven Texture
## 531                    Murad  ...           Dark Spots, Dullness, and Uneven Texture
##
## [532 rows x 7 columns]
```

```
df_recommend2[df_recommend2['Product Description'].isna()] # final N/A fix
```

```
## Empty DataFrame
## Columns: [Brand Name, Product Name, Product Category, Product Description, Product Ingredients, Skin
## Index: []
```

```
import string
from sklearn.metrics.pairwise import cosine_similarity

# function to remove punctuation from columns
def remove_punctuation(value):
    return value.translate(str.maketrans('', '', string.punctuation))
```

```
# df_recommend2 = df_recommend2.astype(str) # make each column a string
df_recommend2['string'] = df_recommend2['Brand Name'].map(remove_punctuation) + " " + \
df_recommend2['Product Name'].map(remove_punctuation) + " " + \
df_recommend2['Product Category'].map(remove_punctuation) + " " + \
df_recommend2['Product Description'].map(remove_punctuation) + " " + \
df_recommend2['Product Ingredients'].map(remove_punctuation) + " " + \
df_recommend2['Skin Type'].map(remove_punctuation) + " " + \
df_recommend2['Skin Concerns'].map(remove_punctuation)
```

```
## <string>:3: SettingWithCopyWarning:
## A value is trying to be set on a copy of a slice from a DataFrame.
## Try using .loc[row_indexer,col_indexer] = value instead
##
## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexir
```

```
tfidf = TfidfVectorizer()
tfidf_matrix = tfidf.fit_transform(df_recommend2['string'])
```

2. **Similarity Calculation**: We computed pairwise cosine similarity scores between products.

```
cosine_similarity = cosine_similarity(tfidf_matrix)

similarity_df = pd.DataFrame(cosine_similarity, index = df_recommend2['Product Name'], columns = df_rec

similarity_df.head()
```

```
## Product Name                                        Cleanser  ...  Mini Rapid Dark Spot Correcting S
## Product Name                                                  ...
## Cleanser                                           1.000000  ...                              0.12
## Aquarius BHA + Blue Tansy Clarifying Cleanser      0.180169  ...                              0.06
## Alpha Beta® AHA/BHA Daily Cleansing Gel            0.191638  ...                              0.12
## Equilibrium Rebalancing Cream Cleanser            0.110345  ...                             0.079
## Mini The POREfessional Get Unblocked Makeup-Rem...  0.092545  ...                              0.07
##
## [5 rows x 532 columns]
```

3. **Recommendation Generation**: For a given product, the system returns the top **10** most recommended products based on similarity scores.

```
df_index = pd.Series(df_recommend2.index, index = df_recommend2['Product Name'])
def give_recommendation(product_name):
    product_index = similarity_df.index.get_loc(product_name)
    top_10 = similarity_df.iloc[product_index].sort_values(ascending=False)[1:11]

    print(f"Skin Care Recommendations for customers buying {product_name} :\n")
    print(top_10)

give_recommendation("Mini Fulvic Acid Brightening Cleanser")
```

```
## Skin Care Recommendations for customers buying Mini Fulvic Acid Brightening Cleanser :
##
## Product Name
## Fulvic Acid Brightening Cleanser                                    0.996931
## Fulvic Acid Brightening Cleanser                                    0.996931
## Kakadu Plum Brightening Vitamin C Serum with Hyaluronic Acid        0.247001
## Mini Plum Plump Hyaluronic Acid Moisturizer                         0.215436
```

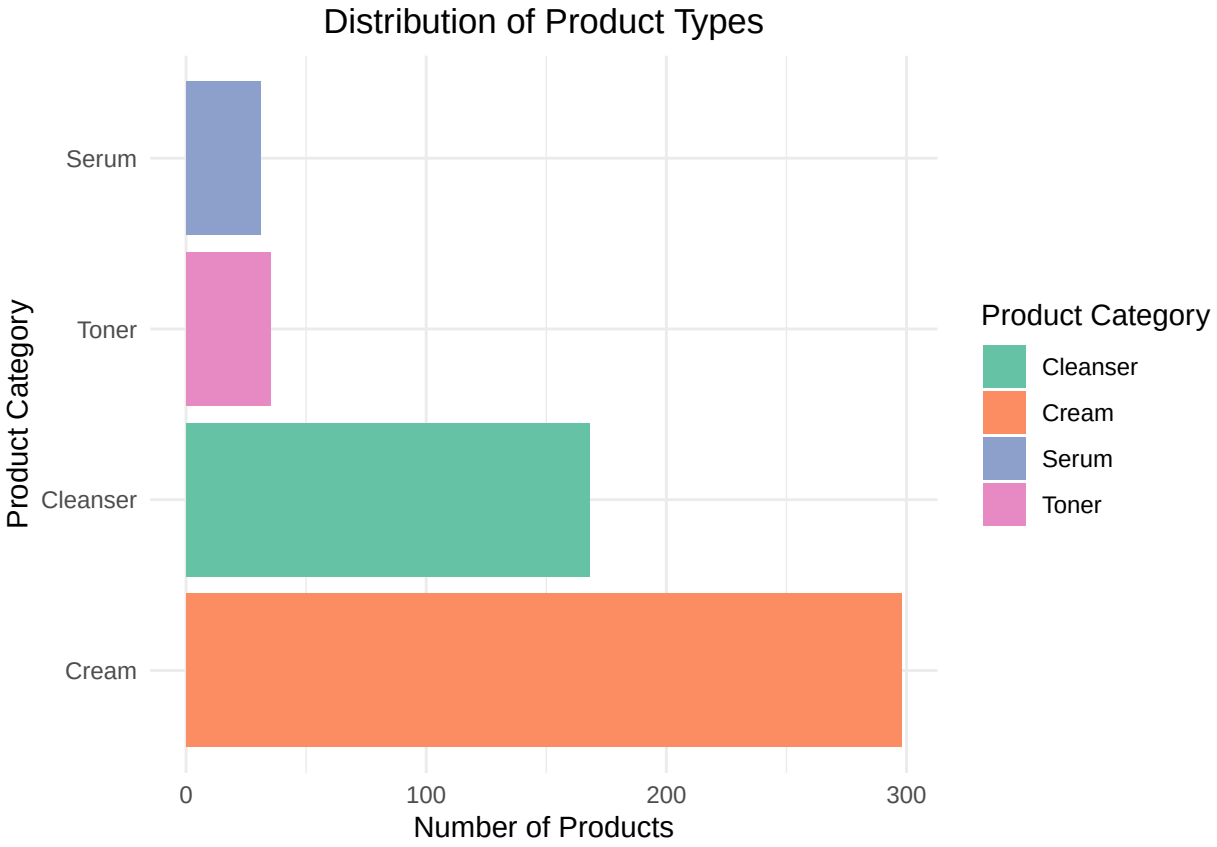
```
## Confidence in a Cleanser Hydrating Facial Cleanser Serum                  0.213414
## Plum Plump Refillable Hyaluronic Acid Moisturizer                         0.211733
## FAB Face Faves - Best of Skincare Cleanse, Exfoliate + Hydrate Gift Set   0.211331
## Vinoclean Gentle Cleansing Almond Milk                                    0.208878
## Vinoclean Makeup Removing Cleansing Oil                                   0.206731
## Gentle Jelly Hydrating Cleanser                                           0.204610
## Name: Mini Fulvic Acid Brightening Cleanser, dtype: float64
```

EDA Visuals

To better understand the composition and characteristics of the skincare product dataset, we visualized key variables such as product category, rating, price, and number of reviews.

Distribution of Product Category

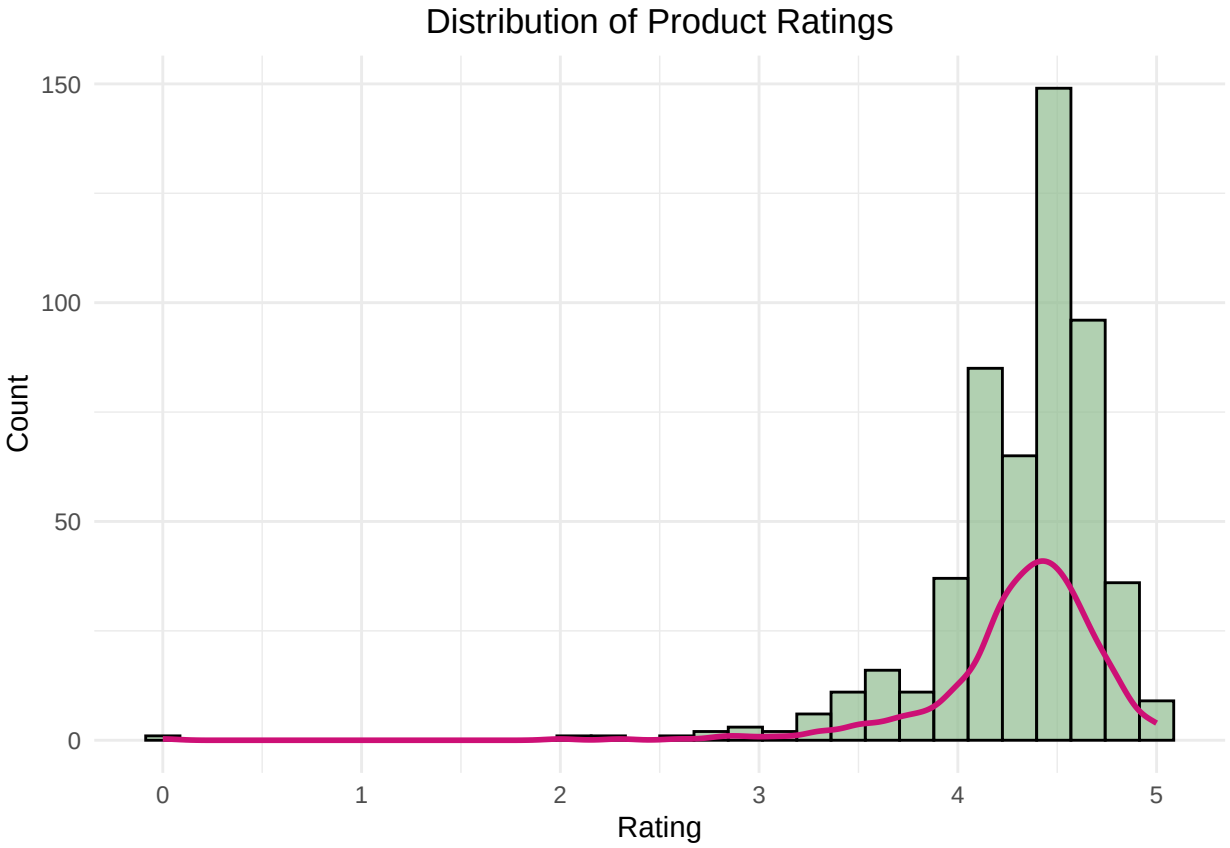
We begin by examining how the products are distributed across different categories. This helps us identify which types of skincare products are most represented in the dataset.



From the chart, we see that certain categories like creams and cleansers are more common over toners and serums. Many users don't have multi-step skincare routines including serum and toner steps due to cost of time and money. Similarly, some skin types are best left alone, and only moisturizer is used which reflects this disproportional bar chart. All in all, users commonly buy and review moisturizers consistently regardless of skin type.

Plot Product Rating and Reviews

Next, we want to look at how product ratings are distributed to understand how satisfied customers are with the available skincare products.

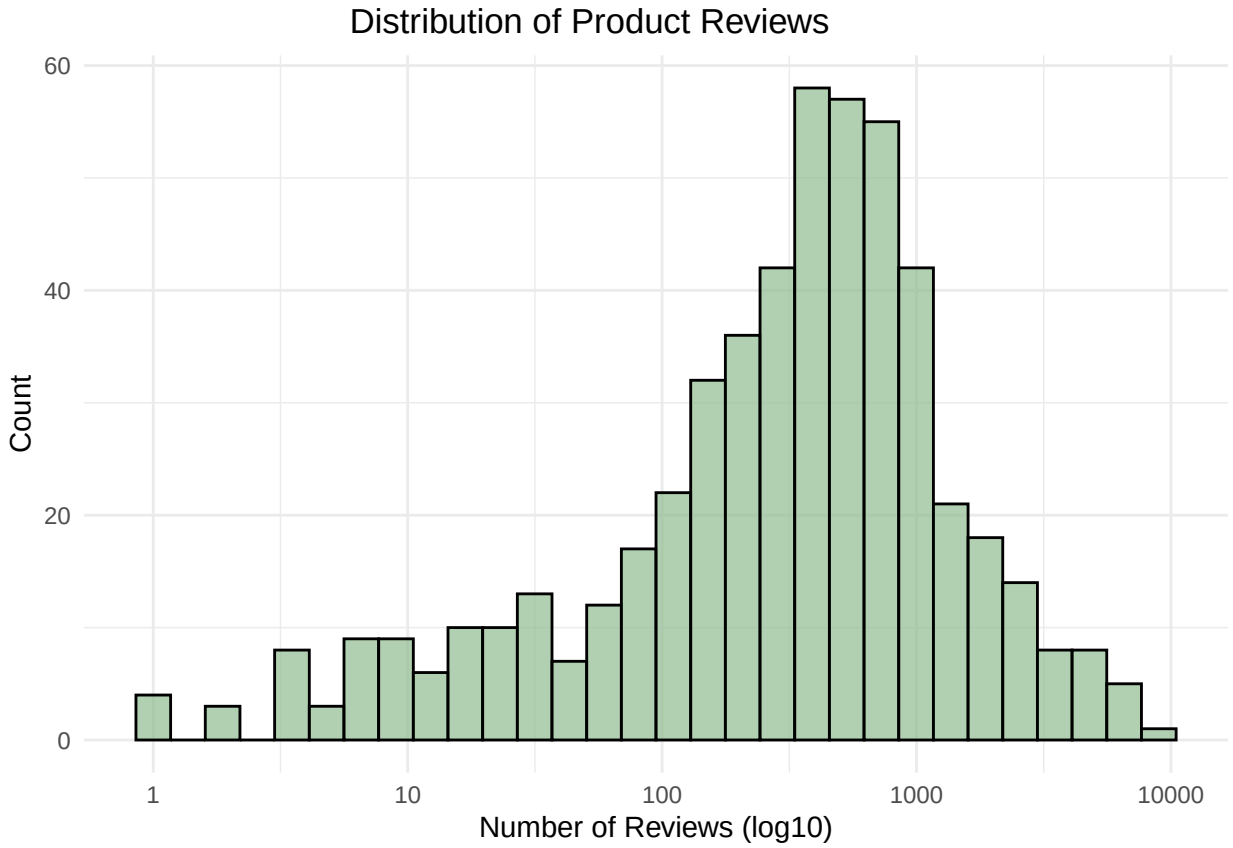


The majority of products have ratings of 4.5, indicating generally positive customer experiences. There's a visible left-skewed distribution, with very few products receiving low ratings. Users that enjoy products could be more likely to leave feedback and repurchase, while those with neutral or negative experiences may disengage without leaving feedback.

If product compatibility (e.g., skin type, concerns) affects satisfaction, then negative or missing reviews might not fully reflect a product's quality but rather a mismatch between user needs and product properties. This is one of the reasons we incorporate skin conditions and ingredients to our model.

Distribution of Product Reviews

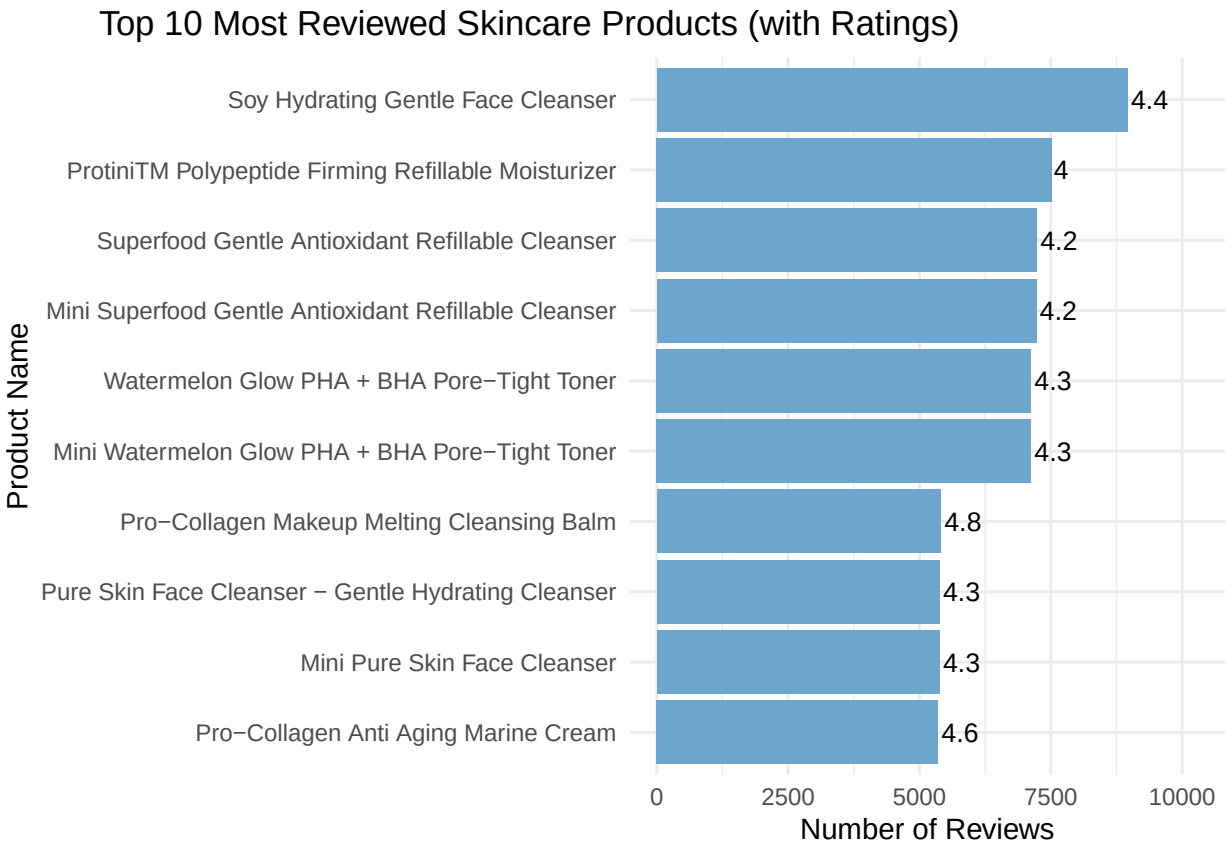
We also want to see how many customer reviews each product recieved. Since review counts can vary widely, we use a logarithmic scale to better represent the spread.



This plot reveals that while most products have fewer than 1000 reviews, a small number of popular products have received many thousands of reviews. Skincare products can gain fame exponentially due to trending brands or ingredients in social media, but this doesn't change compatibility to individual users. We have maximized the recommendation's performance by avoiding bias in review feedback collection—a possible form of self-selection bias.

Top 10 Most Reviews Skincare Products

To identify the most popular products in our dataset, we plotted the top 10 skincare items based on the number of user reviews. In addition to showing popularity, we added each product's average rating as a label to reveal how well these bestsellers are actually received by customers.



We see a wide range of review counts, with all of these products exceeding 5000+ reviews, indicating high visibility and strong customer engagement. Interestingly, not all of the most reviewed products have the highest ratings, highlighting that popularity doesn't always guarantee satisfaction. For example, some top-reviewed products hover around a 4.2–4.4 rating, while others stand out with ratings closer to 4.8 or above.

Conclusion

The web scraping process proved to be both tedious and time-consuming, yet it provided us with a wealth of data—twelve columns' worth of valuable information. Navigating through HTML was akin to deciphering someone else's code, and yet rewarding once finished! Extensive trial and error gained crucial insights that would've saved us time in the long run:

1. Filtering broken links before running the Chromedriver and scraping mass link lists
2. Scraping individual column attributes one at a time for easier debugging and efficient dataframe appending

When deciding between collaborative and content-based recommendations for skincare, we quickly recognized the importance of ingredients, skin concerns, skin types, product brand, and name in determining product compatibility. The challenge of scraping detailed reviews and ratings took considerable effort, but these additional factors proved essential in refining the recommendations. We found that content-based filtering using attributes like ingredients and skin concerns was more suited to our goal of providing personalized skincare suggestions without relying on user-specific data, which would raise privacy concerns. Collaborative filtering, while effective in some contexts, would require additional user data that might compromise privacy and was thus not viable for this project.

Our final system uses the product attributes from Sephora's list—ingredients, skin concerns, and brand

names—to recommend similar skincare products. We also integrated a **rating percentage** to provide additional context for users who might not be as interested in ingredient-based recommendations alone. This hybrid approach balances personalization with respect for privacy, delivering a recommendation system that is both effective and ethical.

Future Directions While our approach was successful, future improvements could focus on enhancing the data quality, refining recommendation accuracy, and incorporating user feedback. Further exploration into hybrid recommendation models, as well as exploring sentiment analysis of reviews, could lead to more personalized and robust results.

Appendix

Web-scraper final loop:

```
from bs4 import BeautifulSoup
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.by import By  # Import By
from selenium.webdriver.common.keys import Keys  # Import Keys for scrolling
import time
import re

#Testing
test_links = categorized_df['link'][1:2]

def scrollDown(driver, n_scroll):
    elem = driver.find_element(By.TAG_NAME, "html")
    while n_scroll >= 0:
        elem.send_keys(Keys.PAGE_DOWN)
        n_scroll -= 1
    return driver

# Setup Chrome options
options = Options()
options.add_argument("--disable-gpu")

#Christina's agent
# options.add_argument(
#   "user-agent=Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
# (KHTML, like Gecko) Chrome/114.0.5735.90 Safari/537.36")
options.add_argument("--disable-blink-features=AutomationControlled")

# Initialize the WebDriver
driver = webdriver.Chrome(options=options)  # Ensure ChromeDriver installed, in PATH

# Loop through each link in categorized_df['link']
products_list = []
for link in test_links:
    try:
        driver.get(link)
        time.sleep(10)  # Give page time to load
```

```
        #Check link if the page redirects to "productnotcarried"
        if "/search?" in driver.current_url:
          print(f" Skipping unavailable product: {link}")
          continue

        while True:
            browser = scrollDown(driver, 20) #scroll down the page
            time.sleep(10) #give it time to load
            break

        #Parse Page Source
        soup = BeautifulSoup(driver.page_source, 'html.parser')

        # Extract product name
        prod_name_element = soup.find('span', {'data-at':
          'product_name'})  # Use find instead of find_all

        if prod_name_element:
          prod_name = prod_name_element.text.strip()  # Extract text
          prod_name = " ".join(
            word for word in prod_name.split() if word.lower() != "hair")
        else:
          prod_name = "N/A"  # Default value if not found

        # Extract brand name
        prod_element = soup.find('a', class_=['css-1kj9pbo e15t7owz0',
                                              'css-wkag1e e15t7owz0'])
        brand_name = prod_element.text.strip() if prod_element else "N/A"

        #Extract brand size
        size = soup.find(['span', 'div'], class_ = ['css-15ro776',
                                                    'css-1wc0aja e15t7owz0'])
        if size:
          prod_size = size.text.strip()   # Extract text
          prod_size = prod_size.replace(
            'Size:', '').replace('Size', '').strip()  # Remove "Size", extra details
        else:
          prod_size = "N/A"  # Default if not found

        #Product type
        category = categorized_df.loc[
          categorized_df['link'] == link, 'category'].values
        category = category[0] if len(category) > 0 else "N/A"

        #Extract product price
        price = soup.find('b', class_='css-0')
        prod_price = price.text.strip() if price else "N/A"

        #Extract product rating
        rating = soup.find_all('span', class_ = 'css-egw4ri e15t7owz0')
        if rating and len(rating) > 0:
          prod_rating = rating[0].text.strip()  # Get first match
        else:
```

```
        ratings_section = soup.find('h2', {'data-at': 'ratings_reviews_section'})
        if ratings_section and "(0)" in ratings_section.text:
          prod_rating = "0"
        else:
          prod_rating = "N/A"

      #Extract brand reviews
      review = soup.find_all('span', class_ = 'css-1dae9ku e15t7owz0')
      if review and len(review) > 0:
        prod_reviews = review[0].text.strip()  # Get full text
        prod_reviews = prod_reviews.replace(",", "")  # if commas remove
        match = re.search(r'\d+', prod_reviews)  # Extract only first number
        prod_reviews = match.group(0) if match else "N/A"  # Get matched number
      else:
        # Check for "Ratings & Reviews (0)" when no reviews exist
        ratings_section = soup.find('h2', {'data-at':
                                'ratings_reviews_section'})
        if ratings_section and "(0)" in ratings_section.text:
          prod_reviews = "0"
        else:
          prod_reviews = "N/A"

      #Extract Description
      time.sleep(3)
      # Locate all divs that may contain product descriptions
      description_classes = ['css-1v2oqzv e15t7owz0',
                             'css-1j9v5fd e15t7owz0',
                             'css-1uzy5bx e15t7owz0',
                             'css-12cvig4 e15t7owz0',
                             'css-eccfzi e15t7owz0',
                             'css-11gp14a e15t7owz0',
                             'css-2f6kh5 e15t7owz0']

      description_tags = ['p', 'b', 'strong']

      # Initialize default value
      prod_desc = "N/A"

      for class_name in description_classes:
        divs = soup.find_all('div', class_=class_name)

        for div in divs:
          for tag in description_tags:
            element = div.find(tag, string=lambda text:
                               text and "What it is:" in text)
            # element = div.find(tag)
            # Check if the element contains "What it is:"
            if element:

              # Extract text while handling possible formatting issues
              extracted_text = element.get_text(separator=" ",
              strip=True).replace("What it is:", "").strip()
```

```
            # Case 2: The description follows the tag as a sibling text
            if not extracted_text and element.next_sibling:
              extracted_text = element.next_sibling.strip()

            # Case 3: The description is inside a `<p>` tag after
            # the strong/b tag if not extracted_text:
            if not extracted_text:
              next_container = element.find_next_sibling("p")
              if next_container:
                extracted_text = next_container.get_text(strip=True)

            # **Case 4: The description is inside a `<div>` right after**
            if not extracted_text:
              next_div = element.find_next_sibling("div")
              if next_div:
                extracted_text = next_div.get_text(strip=True)

            # If valid text is found, set it and stop searching
            if extracted_text:
              prod_desc = extracted_text
              break # Stop searching once we find a valid description

        if prod_desc != "N/A":
          break  # Stop checking other divs once we get correct description

    # Print the extracted product description
    print(f"Product Description: {prod_desc}")

    #Extract Skin Types
    description_classes2 = ['css-1v2oqzv e15t7owz0',
                            'css-1j9v5fd e15t7owz0',
                            'css-1uzy5bx e15t7owz0',
                            'css-12cvig4 e15t7owz0',
                            'css-eccfzi e15t7owz0',
                            'css-11gp14a e15t7owz0',
                            'css-2f6kh5 e15t7owz0']

    description_tags2 = ['p', 'b', 'strong']

    # Initialize default value
    skin_type = "N/A"

    for class_name in description_classes2:
      divs = soup.find_all('div', class_=class_name)

      for div in divs:
        for tag in description_tags2:
          element = div.find(tag, string=lambda text: text and
          ("Skin Type:" in text or "Skin Types:"
          in text or "Skincare Type:" in text or "Skincare Types:" in text))
          if element:

            # Extract text while handling possible formatting issues
```

```
            extracted_text = element.get_text(
              separator=" ", strip=True).replace(
                "Skincare Types:", "").replace(
                  "Skincare Type:", "").replace(
                    "Skin Type:", "").replace(
                      "Skin Types:","").strip()

            # Case 2: The description follows the tag as a sibling text
            if not extracted_text and element.next_sibling:
              extracted_text = element.next_sibling.strip()

            # Case 3: The description is inside a `<p>` tag
            # after the strong/b tag if not extracted_text:
            if not extracted_text:
              next_container = element.find_next_sibling("p")
              if next_container:
                extracted_text = next_container.get_text(strip=True)

            # **Case 4: The description is inside a `<div>` right after**
            if not extracted_text:
              next_div = element.find_next_sibling("div")
              if next_div:
                extracted_text = next_div.get_text(strip=True)

            # If valid text is found, set it and stop searching
            if extracted_text:
              skin_type = extracted_text
              break # Stop searching once we find a valid description

        if skin_type != "N/A":
          break  # Stop checking other divs once we get correct description

    #Extract Concerns
    description_classes3 = ['css-1v2oqzv e15t7owz0',
    'css-1j9v5fd e15t7owz0', 'css-1uzy5bx e15t7owz0',
    'css-12cvig4 e15t7owz0', 'css-eccfzi e15t7owz0',
    'css-11gp14a e15t7owz0', 'css-2f6kh5 e15t7owz0']

    description_tags3 = ['p', 'b', 'strong']

    # Initialize default value
    skin_concerns = "N/A"

    for class_name in description_classes3:
      divs = soup.find_all('div', class_=class_name)

      for div in divs:
        for tag in description_tags3:
          element = div.find(tag, string=lambda text:
            text and ("Skincare Concerns:" in text or
            "Skincare Concern:" in text))
          if element:
```

```
            # Extract text while handling possible formatting issues
            extracted_text = element.get_text(
              separator=" ", strip=True).replace(
                "Skincare Concern:", "").replace(
                  "Skincare Concerns:", "").replace("- ", "").strip()

            # Case 2: The description follows the tag as a sibling text
            if not extracted_text and element.next_sibling:
              extracted_text = element.next_sibling.strip()

            # Case 3: The description is inside a `<p>`
            # tag after the strong/b tag if not extracted_text:
            if not extracted_text:
              next_container = element.find_next_sibling("p")
              if next_container:
                extracted_text = next_container.get_text(strip=True)

            # **Case 4: The description is inside a `<div>` right after**
            if not extracted_text:
              next_div = element.find_next_sibling("div")
              if next_div:
                extracted_text = next_div.get_text(strip=True)

            # If valid text is found, set it and stop searching
            if extracted_text:
              skin_concerns = extracted_text
              break # Stop searching once we find a valid description

        if skin_concerns != "N/A":
          break  # Stop checking other divs once we get correct description

    #Extract Ingredients
    ingredient_element = soup.find('div',
    class_ = 'css-1mb29v0 e15t7owz0')
    if ingredient_element:
      prod_ingredients = ingredient_element.text.strip()
    else:
      prod_ingredients = 'N/A'

    # Append data
    products_list.append({"Brand Name": brand_name,
                          "Product Name": prod_name,
                          "Product Category": category,
                          "Product Price": prod_price,
                          "Product Rating": prod_rating,
                          "Product Size": prod_size,
                          "Product Reviews": prod_reviews,
                          "Product Description": prod_desc,
                          "Product Ingredients": prod_ingredients,
                          "Skin Type": skin_type,
                          "Skin Concerns": skin_concerns,
                          "URL": link})
    #check if it processes link
```

```
        print(f" processed: {link}")

    except Exception as e:
        print(f" Error processing {link}: {e}")

# Close WebDriver *after* processing all links
driver.quit()

product_df = pd.DataFrame(products_list)
print(product_df)
View(product_df)
```