Code

Sephora: Curated Skincare

Content-Based Filtering Recommendation System

Sofia Ward / Bowie Chuang / Christina Pham / Carter Kulm

2025-03-19

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! Misleading advertisements and fake reviews help corporations achieve profit goals while neglecting customers' best interests: 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 curated products** to their specific needs. For user accessibility, we chose Sephora's Skincare selection.

Web-scraped Data Description

To build our system, we collected product data via web scraping from Sephora's website. After checking for permissions, we used the `robots.txt` file to acquire an html of product links (https://www.sephora.com/sitemaps/products-sitemap_en-CA.xml), utilizing `Beautiful Soup` and `Selenium`'s Webdriver. Filtering cosmetics and hair products by key word left us with a `categorized_links` 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.

Show

```
## [1] 799  12
```

Altogether we scraped **1,888 links** which filtered to almost **800 observations and 12 columns.** The final scraping loop is provided in the appendix of this report.

Variable Name	Variable Type	Description
`Product Name`	Character	The name of the skincare product.
`Product Brand`	Character	The company or brand that manufactures the product.

Variable Name	Variable Type	Description
`Product Category`	Character	The type of product: exfoliates, cleansers, toners, serums, moisturizers, masks.
`Product Price`	Double	The cost of the product in USD.
`Product Rating`	Double	The star rating out of 5 for the product.
`Product Reviews`	Double	The number of user reviews for the product.
`Product Size`	Character	The quantity of the product (e.g., 100ml, 1.7oz).
`Product Ingredients`	Character	A list of active and inactive ingredients.
`Product Description`	Character	A textual summary of the product, often provided by the brand or Sephora.
`Skin Concern`	Character	The specific skin issues the product addresses (e.g., acne, dryness, hyper-pigmentation).
`Skin Type`	Character	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.
2. **Feature Engineering**: Formatted price, rating, and size for consistency.

Show

Product Price <dbl>	Product Rating <dbl>	Product Reviews <dbl>
108	4.1	74
26	4.6	269
39	4.3	486
52	4.4	58
18	4.6	498
7	4.3	448

6 rows

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 project, we used cosine similarity and the sigmoid kernel to generate personalized product recommendations which allows us to explore how different similarity metrics impact recommendation results.

Cosine Similarity Content-Based Recommender System

For the *cosine similarity* based recommender system, 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).

Hide

```
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)

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

tfidf_matrix_dense = tfidf_matrix.todense()  # Convert sparse matrix to dense matrix
tfidf_matrix_dense;
```

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

Hide

```
cosine_similarity = cosine_similarity(tfidf_matrix)

similarity_df = pd.DataFrame(cosine_similarity,
                index = df_recommend2['Product Name'],
                columns = df_recommend2['Product Name']
                )
# similarity_df.iloc[2:6, 2:6]
```

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

Hide

```
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].res
et_index()
    top_10.columns = ['Product Name', "Similarity Rating"]
    top_10 = top_10.merge(df_recommend2[['Product Name', 'Product Rating']],
    on="Product Name", how="left")
    print(f"Recommendations for customers buying {product_name} :\n")
    return top_10.style.set_properties(
    **{"background-color": "white","color":"black",
    "border": "1.5px solid black"})

give_recommendation("Vinoclean Gentle Cleansing Almond Milk")
```

	Product Name	Similarity Rating	Product Rating
0	Vinoclean Makeup Removing Cleansing Oil	0.456462	4.200000
1	Nourishing Whipped Almond Delicious Shower Body Cleanser	0.314536	4.700000
2	Brighter Days Ahead Cleansing Trio: Cleansing Balm Starter Set	0.306738	3.300000
3	Nourishing Makeup Removing Oil Cleanser with Squalene and Vitamin E	0.301921	4.400000
4	Mini Oat Cleansing Balm	0.297924	3.700000
5	Midnight Ritual Retinol Renewal Serum	0.297349	4.600000
6	Oat Makeup Removing Cleansing Balm	0.289798	3.700000
7	Barrier+ Lipid-Boost Body Cream	0.280301	4.600000
8	Pro-Collagen Makeup Melting Cleansing Balm	0.279475	4.800000
9	Goat Milk Moisturizing Cream	0.279152	4.000000

Sigmoid Kernel Content-Based Recommender System

Next, we implemented another recommender system using the *sigmoid kernel*. This method transforms the similarity computation through a nonlinear function, allowing it to detect nuanced connections that may not be captured by linear measures like cosine similarity.

$$S_K(x, B) = \tanh(\alpha(x^T \cdot B) + c)$$

Variable Name	Variable Type	Description
`x`	Feature Vector	The feature vector representing the input product.
`tanh`	Mathematical Function	The hyperbolic tangent function.
α	Scalar	A scaling parameter that adjusts how sensitive the kernel is to the similarity score.
x^T	Feature Vector	The transpose of the feature vector representing the input product.
`B`	Feature Vector	The feature vector of another product in the dataset that we want to compare with `x`.
$x^T \cdot y$	Dot Product	The dot product between the two vectors.

Variable Name	Variable Type	Description
c	Scalar	A bias term that shifts the output of the kernel function.

1. **Text Vectorization**: We use the same text vectorization as our cosine similarity recommender system.
2. **Similarity Calculation**: We compute the pairwise similarities between all items in the `tfv_matrix`, resulting in a similarity matrix called `sig` (denoted as `S`). This is a square matrix where both the rows and columns correspond to items—in this case, product names—and each cell (`i`, `j`) represents the similarity score between item `i` and item `j` based on the sigmoid kernel. To facilitate lookup functionality, we use the `Series()` function to map product names to their corresponding indices.

Hide

```
sig = sigmoid_kernel(tfidf_matrix, tfidf_matrix)
df_index = pd.Series(df_recommend2.index, index = df_recommend2['Product Name'])
def give_recommendation2(product_name, sig = sig):
    idx = df_index[product_name]
    sig_score = list(enumerate(sig[idx]))
    sig_score = sorted(sig_score, key = lambda x: x[1], reverse = True)
    # start at 1 to avoid the diagonal/same comparison
    sig_score = sig_score[1:11]
    product_indices = [i[0] for i in sig_score]

    rec_dic = {"No" : range(1,11),
               "Product Name" :
                 df_recommend2["Product Name"].iloc[product_indices].values,
               "Product Rating":
                 df_recommend2["Product Rating"].iloc[product_indices].values}
    dataframe = pd.DataFrame(data = rec_dic)
    dataframe.set_index("No", inplace = True)
    dataframe['Product Rating'] = dataframe['Product Rating'].round(2)

    print(f"Skin Care Recommendations for customers buying {product_name} :\n")
    return dataframe.style.set_properties(
    **{"background-color": "white","color":"black","border": "1.5px  solid black"})
```

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

Hide

```
give_recommendation2("Vinoclean Gentle Cleansing Almond Milk")
```

No	Product Name	Product Rating
1	Vinoclean Makeup Removing Cleansing Oil	4.200000
2	Nourishing Whipped Almond Delicious Shower Body Cleanser	4.700000
3	Brighter Days Ahead Cleansing Trio: Cleansing Balm Starter Set	3.300000
4	Nourishing Makeup Removing Oil Cleanser with Squalene and Vitamin E	4.400000
5	Mini Oat Cleansing Balm	3.700000
6	Midnight Ritual Retinol Renewal Serum	4.600000
7	Oat Makeup Removing Cleansing Balm	3.700000
8	Barrier+ Lipid-Boost Body Cream	4.600000

No	Product Name	Product Rating
9	Pro-Collagen Makeup Melting Cleansing Balm	4.800000
10	Goat Milk Moisturizing Cream	4.000000

Cosine Similarity Content-Based Recommender System with Added Weight

1. **Text Vectorization**: We implemented the same process as our cosine similarity recommender system, converting textual data (e.g., ingredients, descriptions) into vectors using TF-IDF. In addition, **we added weight to** `Brand Name` **and** `Product Category` for a fine-tuned similarity calculation.

Hide

```
df_recommend2['string2'] = (df_recommend2['Brand Name'].map(remove_punctuation) + "
")*4 + \
df_recommend2['Product Name'].map(remove_punctuation) + " " + \
(df_recommend2['Product Category'].map(remove_punctuation) + " ")*4 + \
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)

tfidf2 = TfidfVectorizer()
tfidf_matrix2 = tfidf2.fit_transform(df_recommend2['string2'])
```

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

Hide

```
import pandas as pd
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
cosine_similarity2 = cosine_similarity(tfidf_matrix2)
similarity_df2 = pd.DataFrame(cosine_similarity2,
                index = df_recommend2['Product Name'],
                columns = df_recommend2['Product Name']
                )
# similarity_df2.head()
```

3. **Recommendation Generation**: Just like the previous two, for any given product, our recommender system returns the top **10** most recommended products based on similarity scores.

Hide

```
def give_recommendation3(product_name):
    product_index = similarity_df2.index.get_loc(product_name)
    top_10 = similarity_df2.iloc[product_index].sort_values(ascending=False)[1:11].re
set_index()
    top_10.columns = ['Product Name', "Similarity Rating"]
    top_10 = top_10.merge(df_recommend2[['Product Name', 'Product Rating']],
    on="Product Name", how="left")
    print(f"Recommendations for customers buying {product_name} :\n")
    return top_10.style.set_properties(
    **{"background-color": "white","color":"black","border": "1.5px solid black"})

give_recommendation3("Vinoclean Gentle Cleansing Almond Milk")
```

	Product Name	Similarity Rating	Product Rating
0	Vinoclean Makeup Removing Cleansing Oil	0.562364	4.200000
1	Vinoclean Gentle Foam Cleanser	0.449119	4.200000
2	Vinoclean Cleansing Micellar Water	0.365350	4.000000
3	VinoHydra Deep Hydration Moisturizer	0.352388	4.500000
4	Vinopure Pore Purifying Gel Cleanser	0.352291	4.400000
5	Premier Cru Anti Aging Cream Moisturizer with Hyaluronic Acid	0.334393	4.700000
6	Premier Cru Skin Barrier Rich Moisturizer with Bio-Ceramides	0.329265	4.700000
7	Deep Exfoliating Cleanser	0.323002	4.100000
8	Vinopure Oil-Control Moisturizer for Acne Prone Skin	0.317549	4.300000
9	VinoHydra Moisturizing Mask	0.316662	4.700000

Comparing Sigmoid Kernel vs. Cosine Similarity Approaches

We have tested three different recommender system approaches for skincare product recommendations, each with distinct methodologies and results.

Cosine Similarity with TF-IDF

- Top similarity score: 0.456%
- Best-rated recommendation: 4.2 out of 5 stars

Sigmoid Kernel Similarity

- Best-rated recommendation: 4.2 out of 5 stars
- Similarity score not shown for Sigmoid Kernel Approach
- Recommend the same products in the same order as Cosine Similarity Approach

Cosine Similarity with Weight Adjustment (Product Category & Brand Emphasis)

- Top similarity score: 0.56%
- Best-rated recommendation: 4.2 out of 5 stars
- Most effective method, as it improved similarity rankings while maintaining high-rated recommendations.

*Among the three models, **Cosine Similarity with Weight Adjustment** proved to be the most effective. By emphasizing product category and brand names, this approach significantly improved similarity scores, aligning better with user expectations.*

- Product category weighting was prioritized since users typically search for a specific type of product (e.g., cleanser or moisturizer) rather than an exact brand or formula.
- Brand weighting was introduced based on the dominance of branding in the skincare industry, particularly in a consumer-driven, brand-loyal market like the U.S.

While weighting is inherently subjective, our choice reflects practical consumer behavior patterns, resulting in a more targeted and relevant recommendation system.

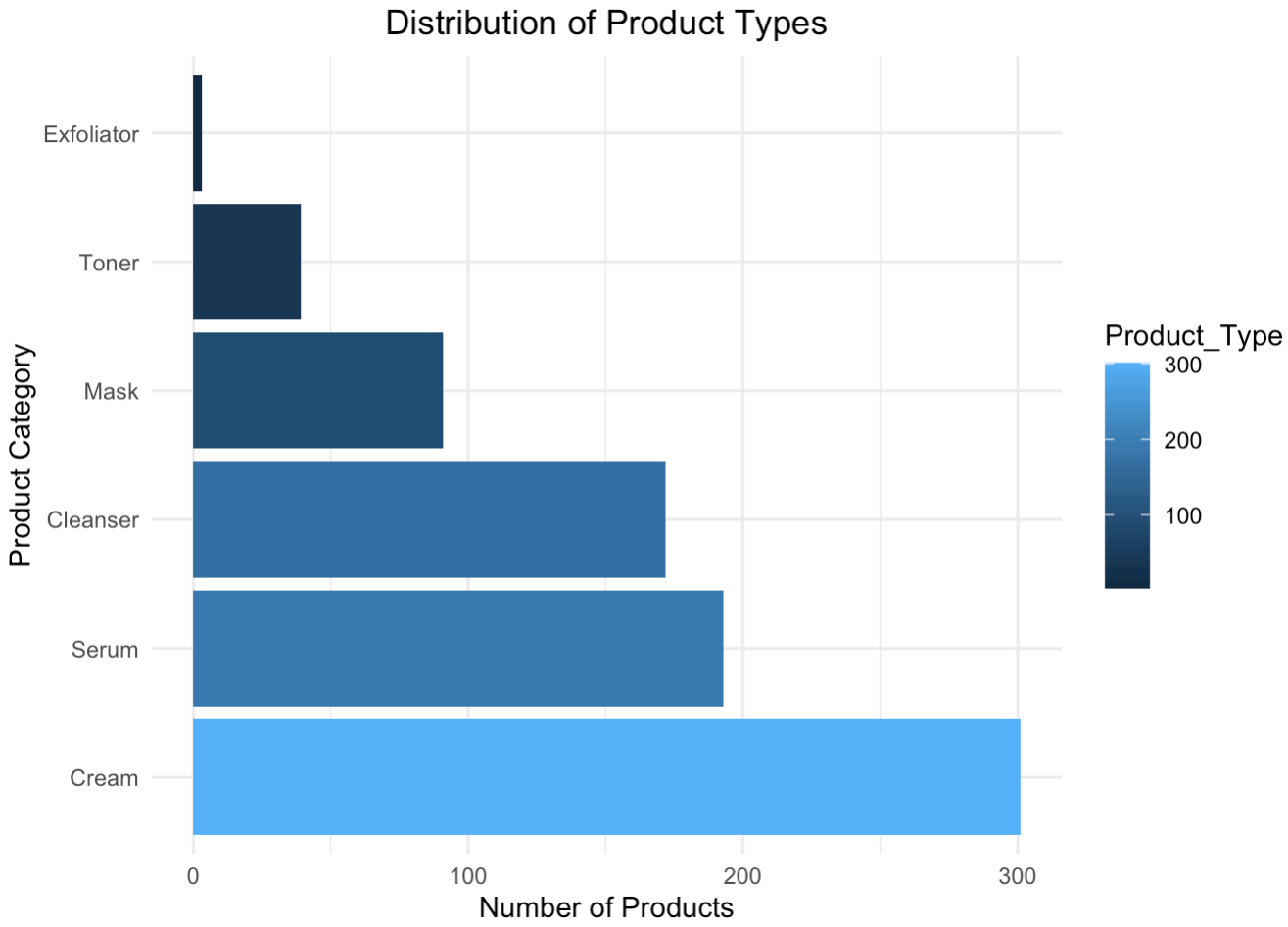
EDA Visuals

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.

Show

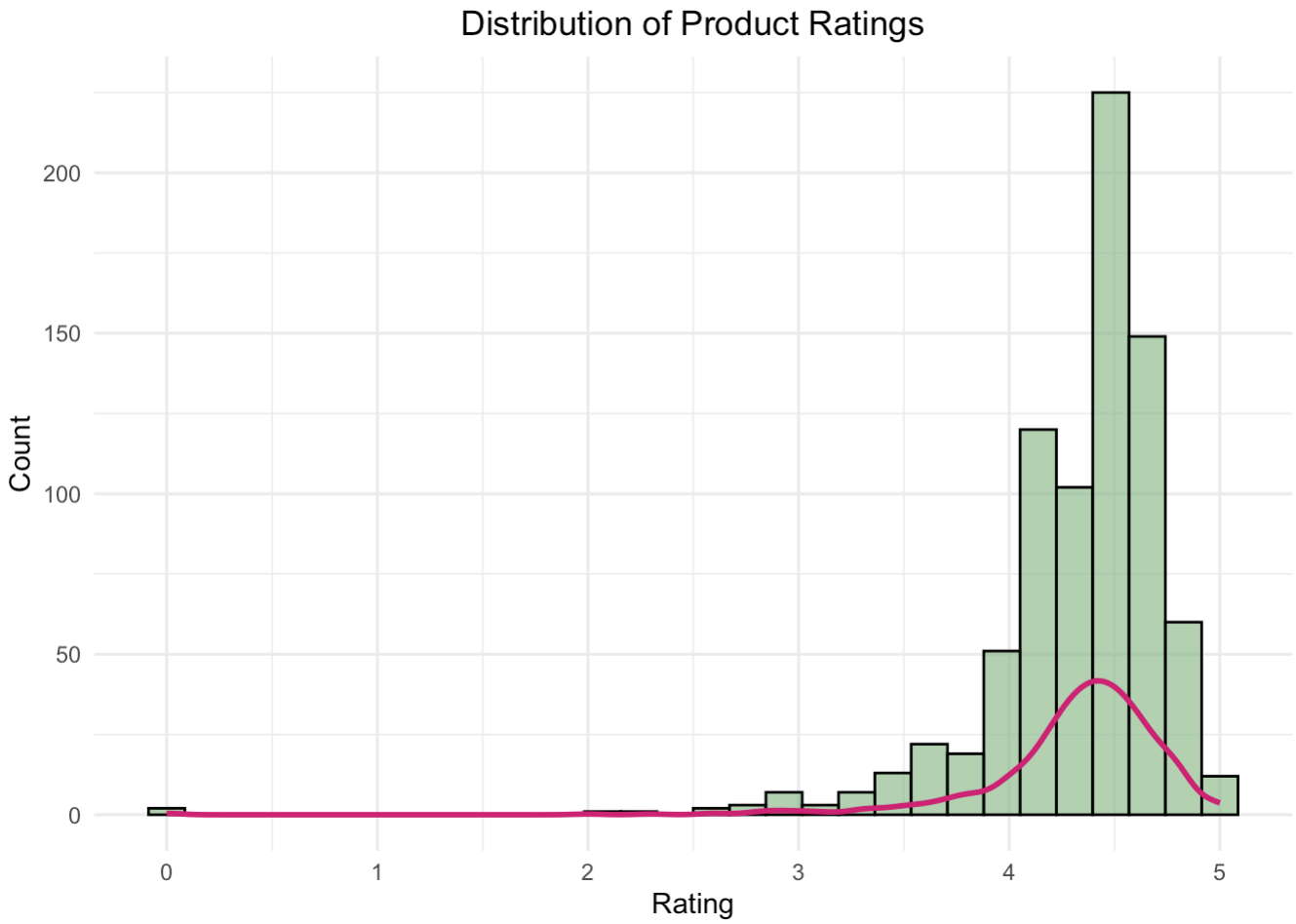


From the chart, we see that certain categories like creams and cleansers are more common over toners and serums. Many users don't have multi-step skincare routines including serum and toner steps due to cost of time and money. Similarly, some skin types are best left alone, and only moisturizer is used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

Show



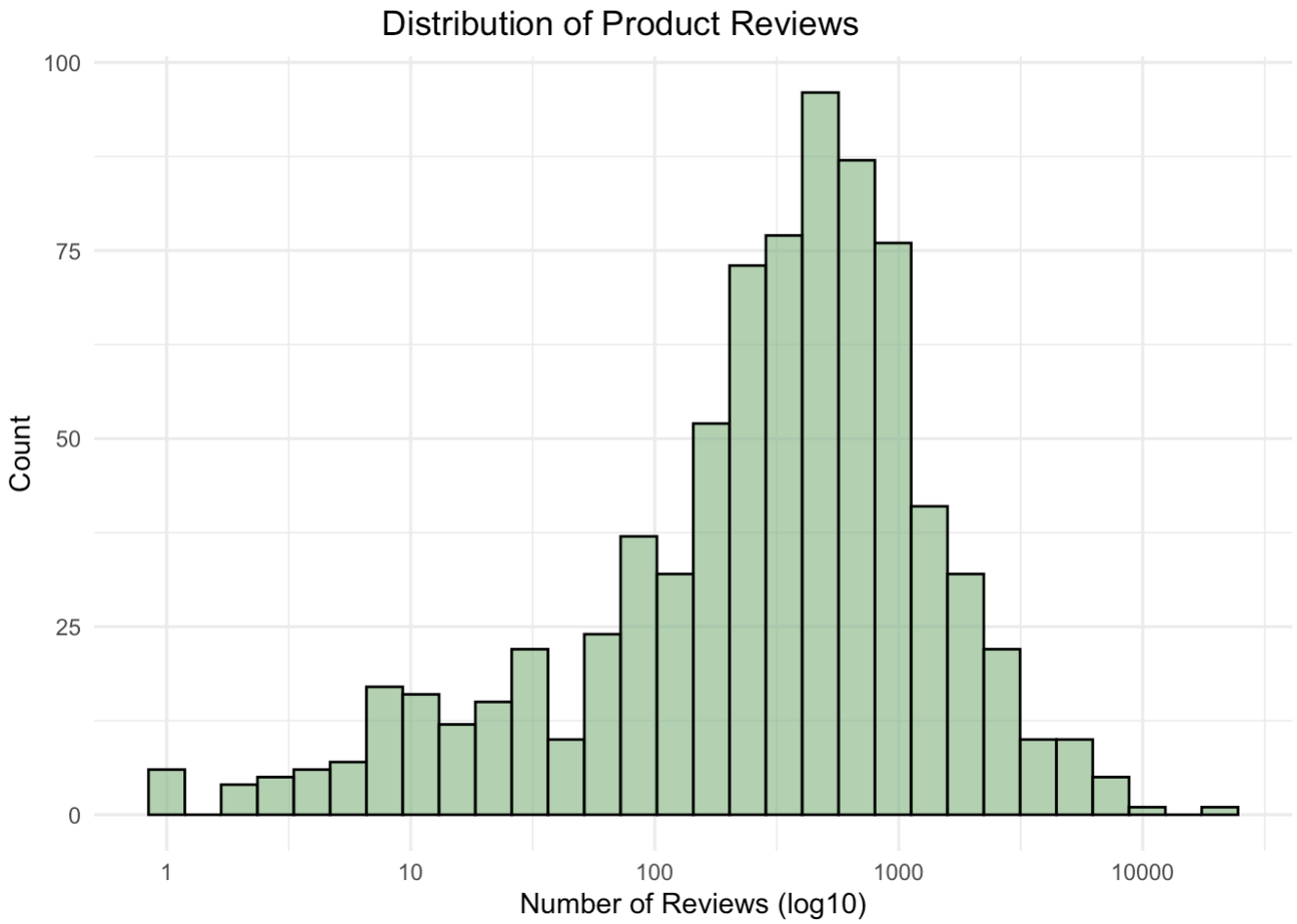
The majority of products have ratings of 4.5, indicating generally positive customer experiences. There's a visible left-skewed distribution, with very few products receiving low ratings. Users that enjoy products could be more likely to leave feedback and repurchase, while those with neutral or negative 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 received. Since review counts can vary widely, we use a logarithmic scale to better represent the spread.

Show

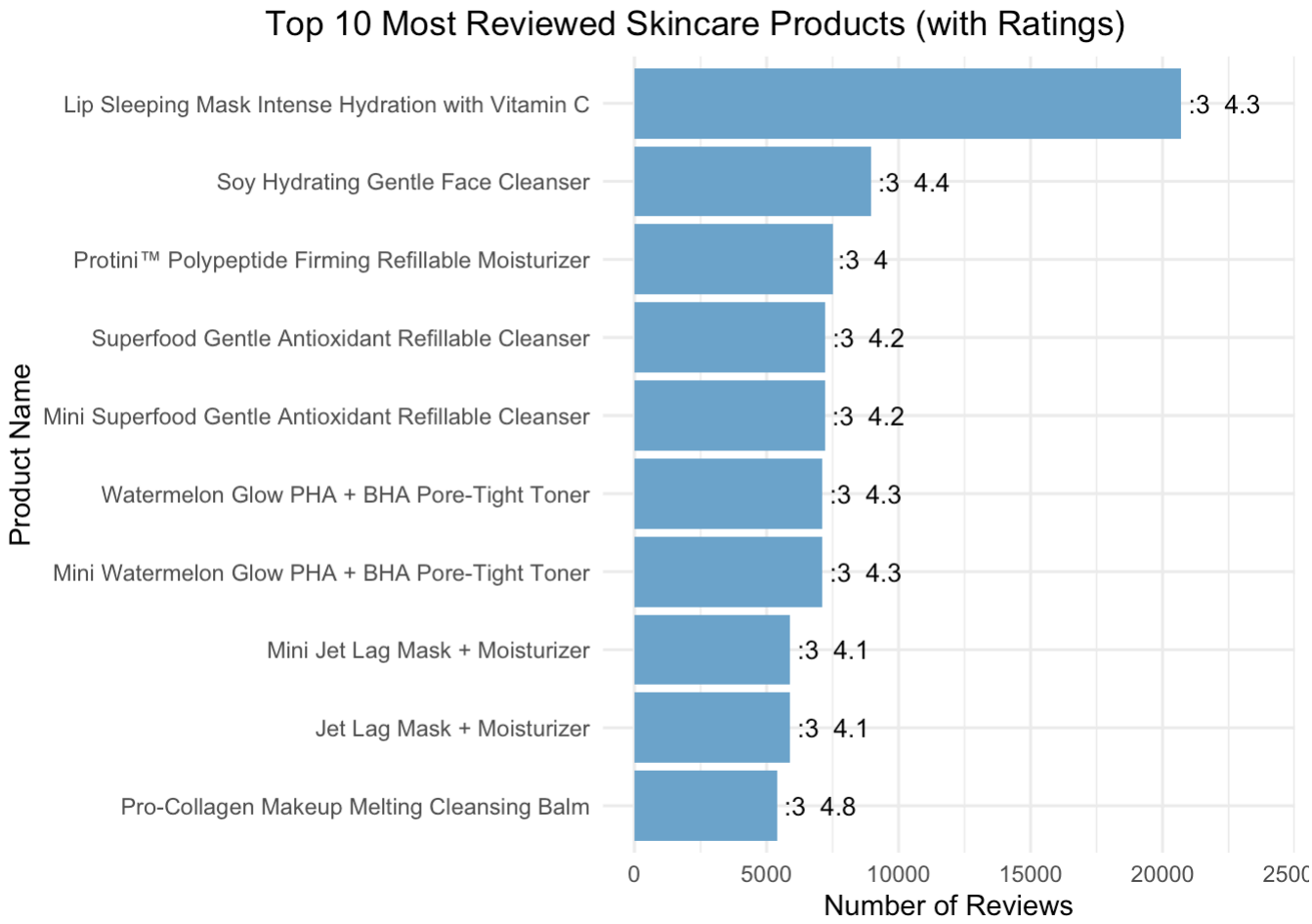


This plot reveals that while most products have fewer than 1000 reviews, a small number of popular products have received many thousands of reviews. Skincare products can gain fame exponentially due to trending brands or ingredients in social media, but this doesn't change compatibility to individual 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.

Show



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 frustrating, and yet rewarding once finished! Extensive trial and error gained crucial insights that would've saved us time in the long run:

- Filtering broken links before running the Chromedriver and scraping mass link lists
- 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 similarity percentage to provide additional context for users who might not be as interested in ingredient-based recommendations alone. This approach balances personalization with respect for privacy, delivering a recommendation system that is both effective and ethical.

Limitations

One notable limitation of our approach is that not all targeted products were successfully scraped. Since we filtered products based on name keywords, some relevant items might have been inadvertently excluded due to inconsistencies in naming conventions or variations in product listings. This constraint may have affected the breadth of our dataset and, consequently, the diversity of recommendations available to users.

Future Directions

While our approach was successful, future improvements could focus on enhancing data quality, refining recommendation accuracy, and incorporating user feedback. Further exploration into hybrid recommendation models, as well as sentiment analysis of reviews, could lead to more personalized and robust results. Additionally, refining our scraping methodology to capture a more comprehensive product list would help ensure a broader and more inclusive recommendation system.

Appendix

Web-scraper final loop:

Hide

```
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)
```