Danmarks Tekniske Universitet

Recommendation System for Skincare Products

02807 Computational Tools for Data Science

Authors

Ana Marija Pavičić - s232468
Anna Sky Kastl Jensen - s194824
Raquel Moleiro Marques - s243636
Sree Keerthi Desu - s243933

December 5, 2024

Contents

Table of Contents

1 Introduction

1.1 Motivation

In today's consumer-driven world, where corporations frequently flood the market with new products and aggressive promotions, choosing the right products can be overwhelming. Our recommendation system offers a simple solution by tailoring suggestions to a user's preferences and predicting potential responses to specific inputs. By presenting users with options they are likely to enjoy, these systems can reduce cognitive overload, making the decision-making process more efficient. As a result, users are more likely to purchase products they genuinely value, ultimately reducing wasteful consumption.

1.2 Approach

Collaborative filtering is a specific type of recommendation system that focuses on the relationships between users and products. Recommendations are generated by identifying similarities between user profiles, which are typically based on their product purchase history [1]. In essence, these recommendations suggest products that other users frequently buy alongside the input product. However, traditional collaborative filtering struggles with the cold start problem, where new users or products lack sufficient data for accurate recommendations.

For our skincare recommendation system, the goal is to provide recommendations based on a singular input skincare product, which simulates the situation where there is a lack of a solid user profile. To achieve this, we first use the A-priori algorithm to analyze product reviews and identify frequent itemsets — groups of products often bought together. This step narrows the search space, focusing on products that are contextually relevant to the user's input.

Next, sentiment analysis is applied to user reviews to gauge overall satisfaction and identify potential adverse reactions. Additionally, we refine the recommendations by calculating similarity scores based on two critical aspects: product highlights (e.g., claims like "brightening" or "hydrating") and ingredients (e.g., hyaluronic acid or salicylic acid). Combining these factors would ensure that the recommended products align with user preferences. By doing so, we aim to answer the following research question:

"How can we deliver product recommendations for customers with no prior user data?"

2 Datasets

The chosen datasets are available on Kaggle under the name "**Sephora Products and Skincare Reviews**" [2]. It consists of a product information file and several files containing different reviews. Please note that all our datasets and code are available in the GitHub repository [3].

2.1 Product Dataset

There are approximately 8,000 products available on Sephora, and their corresponding information includes price, highlights, ingredient list, average rating, size, etc. However, not all of these products are related to skincare. Using the primary category, only skincare products were filtered resulting in 2,420 unique product IDs. Among the remaining rows, there are also mini versions and gift sets made of products that already exist in the dataset and therefore were also removed to ensure that there were no duplicates for the association rules to avoid recommending the same product. Rows that have only either highlights or ingredients were removed, as both columns are necessary for obtaining similar items. This action resulted in 1,583 unique products.

The ingredient list for each product was preprocessed to make sure that ingredients like water (that were listed in multiple different ways such as "aqua") were standardized, and special characters towards the beginning and the end of the list were removed. The uniformity of the column would help the similarity score further in the report.

2.2 Review Dataset

The multiple files containing reviews were merged into one. Most of the products seem to have at least 65 reviews; hence, the review dataset size is considerably large and has over a million reviews. The dataset was reduced to include the reviews corresponding to only the 1583 products obtained from data cleaning of the product dataset. Additionally, the dataset was filtered to keep only the reviews that had 3, 4, or 5 stars as we do not want to consider any negative reviews before finding association rules among them. After all this preprocessing, we end up with 700,000 reviews in the dataset.

3 Methods

3.1 Sentiment Analysis

To ensure that the products we recommend are those that users truly enjoy, we first had to narrow down the data used in our analysis and models. With that in mind, sentiment analysis was implemented as it is a Natural Language Processing (NLP) technique that evaluates the emotional tone of text, classifying it as positive, negative, or sometimes neutral. In the context of product reviews, sentiment analysis helps identify whether users had a positive or negative experience with a product, which is essential when making product recommendations.

Originally, we considered using the *is_recommended* column already present in the reviews dataset, to more easily determine which products users spoke well of. However, we found inconsistencies in this parameter, including a significant amount of missing values and 5-star reviews marked as "not recommended". This made the *is_recommended* column unreliable, making sentiment analysis a necessary alternative.

In our analysis, we focused on reviews rated 3 stars and above, excluding 1- and 2-star reviews as they were considered too negative. Since 4- and 5-star reviews are clearly positive, they were included directly in our study, while sentiment analysis was applied to only 3-star reviews, as their sentiments are ambiguous. To perform the analysis, we used two *Hugging Face* models: *xlnet-base-cased-product-review-sentiment-analysis* [4], specialized for product reviews, and *cardiffnlp/twitter-xlm-roberta-base-sentiment* [5], which supports multiple languages. This allowed us to capture both English and the few non-English reviews. While these models are somewhat of a black box, they are verified, widely used, and well-suited to our problem. To ensure accuracy, we intersected the results of both models, keeping only reviews marked as positive by both. This decreased false positives and therefore resulted in 14,454 positive reviews out of 60,703 3-star reviews for further analysis.

Note that, although there are challenges when taking the sentiment of product reviews such as ambiguous or softened feedback, the implementation of sentiment analysis still plays a crucial role in improving the quality of our recommendations.

3.2 Frequent Itemsets

The primary objective of using frequent itemsets is to uncover associations between products based on customer reviews. Through mining frequent itemsets and generating association rules, we aim to predict which products a customer might like based on their interest in another product. This insight allows us to build a recommendation system that enhances the shopping experience by suggesting items that other similar users have enjoyed. We chose to use the A-priori algorithm because its systematic approach to pruning the search space of itemsets ensures computational efficiency, making it suitable for our dataset size and structure.

Choosing the appropriate minimum support threshold and deciding how many reviews to exclude were critical steps in our analysis. Excluding more users resulted in a larger number of association rules, providing richer data for recommendations. However, this approach risks overfitting to a small subset of users, making the rules less generalizable to the broader population. Conversely, including too many users with fewer reviews requires significantly lowering the minimum support threshold in order to have enough association rules to work with, which can lead to less reliable rules. After exploring various configurations, as can be seen in Table 1, we determined that restricting the dataset to users with 10 or more reviews struck an effective balance. This approach ensured that the generated rules supported meaningful and actionable recommendations.

A limitation of association rule mining is that not all products will have associated rules, especially those with low representation in the dataset. To address this, we incorporated a similar items approach. When a product lacks direct recommendations, we identify the most similar product that is present in the association rules. This ensures every product can act as an entry point for personalized recommendations. A more detailed accounting of this is provided later in the paper.

By mining frequent itemsets and generating association rules, we enable a data-driven recommendation system that suggests products based on user preferences. Furthermore, the integration of similar items enhances its usability across diverse customer journeys.

User Threshold	Association Rules Size	Unique Products Included	Comments
5+ reviews	6	3	Although we are including more users in our dataset, the unique products the resulting rules cover are nowhere near enough to make assumption about the rest of the dataset.
10+ reviews	1704	82	This seems to be a decent middle-ground, where enough unique products are included in the association rules, but we are not cutting out too much of the population.
30+ reviews	526,853	277	Here, we are approaching a super small subset of users. They account for only 0.001% of the total users, which is not enough to make generalizations.

Table 1: Finding a user threshold based on resulting association rules

3.3 Similar Items

Similar items can be found in different ways. Two of the most used measures are Cosine similarity and Jaccard similarity scores. Cosine similarity measures the distance between two vectors in a vector space, whereas Jaccard similarity measures the proportion of shared to the union of the items [1]. As mentioned briefly in our approach, the highlights and ingredients of each product will be used to determine a score. This score is obtained by calculating the similarity scores for highlights and ingredients individually, and then combining them using an approach known as Reciprocal Rank Fusion (RRF).

3.3.1 Similarity of Highlights

The product highlights are sets of the most emphasized features of the product. For example, a certain product targeted at users with dry skin would have a "dry skin" highlight to convey this attribute of the product. To find the similarity of preprocessed highlights between a random product and the input, we computed Jaccard Similarity. It measures how many features are in common within the sets of highlights. The more similar the sets, the more alike the products are in how they affect the skin and address skin issues.

3.3.2 Ingredient Similarity

Before calculating the similarity scores for ingredients, we apply a TF-IDF vectorizer to transform the ingredient lists into vectors. It prevents common ingredients, like water which is a common ingredient in almost all products, from disproportionately influencing similarity, focusing instead on more unique ingredients. The TF-IDF vectorizer accounts for both the frequency of an ingredient within a single product and its frequency across all products. [1]. The resulting vectors are then used to compute cosine similarity scores. The higher the similarity score, the more alike the ingredient lists are in terms of truly relevant ingredients.

3.3.3 Reciprocal Rank Fusion (RRF)

As the similarity scores are generated using different measures, their scales vary. Cosine similarity ranges from -1 to 1, whereas Jaccard similarity ranges from 0 to 1. Therefore, simply taking the average of the scores or any weighted sum would not be recommended.

To avoid this, we use an algorithm known as Reciprocal Rank Fusion (shown in Equation 1),

$$\mathrm{RRF} = \sum_{i=1}^{N} \frac{1}{k + r_i(d)} \tag{1}$$

where $r_i(d)$ is the rank of product d in the i^{th} ranked list, k is a constant (typically $k = 60$) used to prevent division by zero and to adjust the impact of higher ranks, and N is the number of ranked lists (in our case 2) [6].

RRF uses the ranks of the product within the individual lists and then computes a singular score. The rank is simply the position of the product on a list sorted based on the similarity score in descending order. By doing so, we ensure that the higher the score, the more the product is similar to the input in terms of both highlights and ingredients. In the end, the resulting output of RRF is a ranked list of products.

4 Recommendation System

As mentioned briefly in our approach, the recommendation system operates with a logic similar to Collaborative Filtering, but is specifically designed to address the challenge of limited user history. Instead of relying on prior user data, it uses a single product as input, simulating scenarios where no purchase history or profile information is available. To recommend only products that other users have positively reviewed, we first filter reviews with ratings of 4 and 5 stars, as these are inherently positive. For 3-star reviews, we perform sentiment analysis, retaining only those with high sentiment scores, to ensure they reflect a positive experience. The A-priori algorithm is then applied to this filtered data to identify frequent itemsets, revealing which products are commonly bought together. This assumes that users with similar skin conditions tend to purchase and like related items. To handle cases where a product lacks any association rules — such as newly introduced items — the system calculates similarity scores against antecedents in all the found association rules. It identifies the most similar antecedent and utilizes its association rules to generate recommendations. This approach maintains consistency in the type of recommendations provided. Similarly, if there are insufficient association rules, the system uses the similar items method to find more products related to the consequents to ensure a consistent and diverse output.

5 Concluding remarks

One challenge we faced was in determining how many reviews to include in the analysis. Using a smaller subset of reviews often results in more association rules, which provides more granular information for recommendations. However, this approach risks bias, as it may not accurately reflect the broader population's preferences. To address this, two strategies were considered. By plotting user reviews (2+ review, 3+ reviews, etc.) against average confidence of the generated rules, the elbow point in the curve could help us identify a trade-off between rule confidence and dataset size. Another approach would be to use Algorithm 1 as described in the article "Mining association rules with multiple minimum supports" [7], where all the association rules are tested, and only the most influential are used to make assumptions about user preferences. Although we tried to incorporate these methods, we couldn't make it work with the time constraints we had, so we chose arbitrary values for the minimum threshold and user cutoffs. They were chosen based off of what would give decent results without giving over-fitted results. If given the opportunity to improve our recommendation system, we would like to implement these methods.

Another improvement that can be made is to increase the size of the dataset by scraping the data to include more products along with more information about the products themselves such as the product descriptions. The latter would have proved useful in performing similarity score analysis on as it has more textual information. An attempt was made to scrape data, but due to time constraints, we had to pivot.

A Appendix

A.1 Group Member's Contribution to the Project

	Ana Marija Pavičić	Anna Sky Kastl Jensen	Raquel Moleiro Marques	Sree Keerthi Desu
Motivation	40%	20%	0%	40%
Approach	35%	30%	0%	35%
Product Dataset	30%	0%	40%	30%
Review Dataset	20%	0%	60%	20%
Sentiment Analysis	0%	20%	80%	0%
Frequent Itemsets	0%	70%	30%	0%
Similar Items	50%	0%	0%	50%
Recommendation System	35%	20%	10%	35%
Concluding Remarks	15%	70%	0%	15%

Table 2: Contributions Table

A.2 Code

```

# -*- coding: utf-8 -*-
"""final_notebook.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1w5MPyzzZFuzAlRrybLQoq-_WMXtN2hRW

# 02807 Computational Tools for Data Science - Final Project

Group members:
 - Ana Marija P a v i i  (s232468)
 - Anna Sky Kastl Jensen (s194824)
 - Raquel Moleiro Marques (s243636)
 - Sree Keerthi Desu (s243933)

#### Imports
"""

# Commented out IPython magic to ensure Python compatibility.
# %%capture
# !pip install nbimporter
# from imports import *

"""# 1. Data pre-processing"""

# Data paths
raw_data_directory = 'data'
raw_data_path = 'data/product_info.csv'
processed_data_directory = 'processed_data'
processed_data_path = processed_data_directory + '/skincare.csv'

"""## Data Cleaning"""

# Read all products info
df = pd.read_csv(raw_data_path)

# Filter df for rows where 'primary_category' is 'Skincare'
skincare_df = df[df['primary_category'] == 'Skincare']
print("Skin care data size:", len(skincare_df))

# Remove rows where highlight are non existent
skincare_df = skincare_df[skincare_df['highlights'].notna() & (skincare_df['highlights'] != '')]
print("Skin care data size after removing empty highlights:",len(skincare_df))

# Remove rows where ingredients are non existent
skincare_df = skincare_df[skincare_df['ingredients'].notna() & (skincare_df['ingredients'] != '')]
print("Skin care data size after removing empty ingredients:",len(skincare_df))
```

```python

# Remove 'Mini' size products from data as it could be seen as a duplicate
skincare_df = skincare_df[~skincare_df['product_name'].str.contains('mini', case=False, na=False)]
# Also removing additional products with 'Mini Size' as their secondary category
skincare_df = skincare_df[skincare_df['secondary_category'] != 'Mini Size']
print(f"Size of skincare data after removing 'mini' from product names (case-insensitive): {len(skincare_df)}")

# Remove 'Limited edition' from data
skincare_df = skincare_df[~skincare_df['product_name'].str.contains('limited edition', case=False, na=False)]
print(f"Size of skincare data after removing 'limited edition' from product names (case-insensitive): {len(skincare_df)}")

# Remove 'Value & Gift Sets' from data to focus on individual products
skincare_df = skincare_df[skincare_df['secondary_category'] != 'Value & Gift Sets']
print("Skin care data size after removing 'Value & Gift Sets' secondary_category:",len(skincare_df))

# Clean highlight column
highlights = skincare_df['highlights']
highlights = [h.replace("[", "").replace("]", "").replace("'", "").replace("Best for ", "").replace("Good for: ", "").replace(" Skin", "").replace("/", ", ") for h in highlights]
skincare_df['highlights'] = highlights

# Function to clean each ingredient row
def clean_ingredients(row):
    # Replace unwanted characters
    row = row.replace("[", "").replace("]", "").replace("'", "").replace(" (Vegan)*", "").replace(".", "")

    # Remove text inside parentheses
    row = re.sub(r'\([^)]*\)', '', row)

    # Replace " , " with a single comma (in case extra spaces after commas)
    row = row.replace(" ,", ",")
    row = row.replace("  ", " ")

    # Check for "water", "aqua", or "eau" and replace first occurrence
    if "water" in row.lower() or "aqua" in row.lower() or "eau" in row.lower():
        row_list = row.split(", ")
        # Find the first occurrence of "water", "aqua", or "eau"
        index = next((i for i, s in enumerate(row_list) if 'water' in s.lower() or "aqua" in s.lower() or "eau" in s.lower()), -1)
        if index != -1:
            # Replace the identified word with "Water"
            row_list[index] = "Water"
        row = ", ".join(row_list)

    return row

# Apply the clean_ingredients function
skincare_df['ingredients'] = skincare_df['ingredients'].apply(clean_ingredients)

# Ensure the directory exists
output_dir = os.path.dirname(processed_data_path)
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Save cleaned file
skincare_df.to_csv(processed_data_path, index=False)

"""## Clean and Filter Product Reviews"""

# Convert product_id column to a set for efficient filtering
product_ids = set(skincare_df['product_id'])

def clean_reviews(raw_data_path, processed_data_path):
    """
    Clean the reviews in the given CSV file by filtering based on the conditions.
    - Filters rows where 'product_id' is in the skincare dataset.
    - Keeps reviews with rating 5, 4, and only 3-star reviews if they have review text.
    """
    try:
        # Read the data
        df = pd.read_csv(raw_data_path)

        # Keep only the filtered skincare data
        df = df[df['product_id'].isin(product_ids)]

        # # Check for 3-star reviews with no text or empty text
        # three_star_reviews = df[df['rating'] == 3]
        # no_text_reviews = three_star_reviews[three_star_reviews['review_text'].isna() | (three_star_reviews['review_text'].str.strip() == '')]
```

```python

        # Keep reviews with rating 5, 4, or 3 (only keep 3-star reviews if they have review text)
        df = df[
            (df['rating'].isin((5, 4))) |
            ((df['rating'] == 3) & df['review_text'].notna() & (df['review_text'].str.strip() != ""))  # Include 3-star reviews only if they have text
        ]

        # Ensure the output directory exists
        output_dir = os.path.dirname(processed_data_path)
        if not os.path.exists(output_dir):
            os.makedirs(output_dir)

        # Save the cleaned file
        df.to_csv(processed_data_path, index=False)
        print(f"Processed and saved: {processed_data_path}")

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

# Apply the function to each review data file
clean_reviews('data/reviews_0-250.csv', 'processed_data/reviews_0-250.csv')
clean_reviews('data/reviews_250-500.csv', 'processed_data/reviews_250-500.csv')
clean_reviews('data/reviews_500-750.csv', 'processed_data/reviews_500-750.csv')
clean_reviews('data/reviews_750-1250.csv', 'processed_data/reviews_750-1250.csv')
clean_reviews('data/reviews_1250-end.csv', 'processed_data/reviews_1250-end.csv')

print("All files have been processed and saved.")

"""## Combining All Reviews Files Into a Single Dataframe"""

# List of file paths for the processed reviews
file_paths = [
    'processed_data/reviews_0-250.csv',
    'processed_data/reviews_250-500.csv',
    'processed_data/reviews_500-750.csv',
    'processed_data/reviews_750-1250.csv',
    'processed_data/reviews_1250-end.csv'
]

# Combine all files into one DataFrame
combined_reviews_df = pd.DataFrame()

for file_path in file_paths:
    try:
        # Read each processed file with updated argument for handling bad lines
        df = pd.read_csv(file_path,
                         on_bad_lines='skip',  # Skip bad lines instead of using deprecated arguments
                         encoding='utf-8',  # Ensure proper encoding
                         engine='python')  # Use Python engine for more robust handling

        # Append it to the combined DataFrame
        combined_reviews_df = pd.concat([combined_reviews_df, df], ignore_index=True)
        print(f"Successfully added {file_path}")
    except Exception as e:
        print(f"Error reading {file_path}: {e}")

# Save the combined DataFrame to a single CSV file
output_combined_path = 'processed_data/combined_reviews.csv'
combined_reviews_df.to_csv(output_combined_path, index=False)
print(f"All files have been combined and saved to: {output_combined_path}")

"""# 2. Exploratory Data Analysis (EDA)

### Skincare data
"""

# Display the first few rows to get a sense of the data
print("Skincare Data Overview:")
print(skincare_df.head())

# Get general info
print("\nSkincare Data Info:")
print(skincare_df.info())

# Describe numerical columns for basic statistics
print("\nSkincare Data Statistics:")
print(skincare_df.describe())

"""### Product Reviews Data"""

# Display the first few rows to get a sense of the data
print("\nReview Data Overview:")
```

```python
print(combined_reviews_df.head())

# Get general info
print("\nReview Data Info:")
print(combined_reviews_df.info())

# Describe numerical columns for basic statistics
print("\nReview Data Statistics:")
print(combined_reviews_df.describe())

"""### Distribution of categorical and numerical features

#### Skincare data
"""

# Check distribution of secondary categories
print("\nSkincare Secondary Categories Distribution:")
secondary_category_counts = skincare_df['secondary_category'].value_counts().sort_values(ascending=False)
print(secondary_category_counts)

# Plot the distribution of secondary categories
plt.figure(figsize=(10,6))
sns.countplot(
    y='secondary_category',
    data=skincare_df,
    order=secondary_category_counts.index  # Sort the plot by frequency
)
plt.title('Distribution of Secondary Categories')
plt.show()

# Get the count and percentage of top brands
brand_count = skincare_df['brand_name'].value_counts()
brand_percentage = skincare_df['brand_name'].value_counts(normalize=True) * 100

# Create a DataFrame for the top brands
brand_df = pd.DataFrame({'Brand': brand_count.index, 'Counts': brand_count.values, 'Percent': np.round(brand_percentage.values, 2)})

# Display the top 10 brands
print(brand_df.head(10))

# Plot the top 10 brands horizontally
plt.figure(figsize=(8, 6))
sns.barplot(x='Counts', y='Brand', data=brand_df.head(10))
plt.xlabel('Count')
plt.title('Top 10 Occurring Brands')
plt.show()

"""#### Product Reviews Data"""

# Review rating distribution
print("\nReview Ratings Distribution:")
print(combined_reviews_df['rating'].value_counts())

# Plot the rating distribution
plt.figure(figsize=(8,6))
sns.countplot(x='rating', data=combined_reviews_df)
plt.title('Distribution of Review Ratings')
plt.show()

# Review length (characters in review text)
combined_reviews_df['review_length'] = combined_reviews_df['review_text'].apply(lambda x: len(str(x)))

# Plot distribution of review lengths
plt.figure(figsize=(10,6))
sns.histplot(combined_reviews_df['review_length'], bins=50, kde=True)
plt.title('Distribution of Review Lengths')
plt.show()

"""---

## Filtering 3-Star Reviews
"""

# Load the combined reviews data from the saved file
combined_reviews_df = pd.read_csv('processed_data/combined_reviews.csv', encoding='utf-8', engine='python')

# Filter for reviews with rating = 3 to perform sentiment analysis on the same
three_stars = combined_reviews_df[combined_reviews_df['rating'] == 3]

print(f"Total reviews with rating of 3: {len(three_stars)}")

"""## *is_recommended* column in the dataset
```

```
To ensure we would recommend products well-regarded by users, we explored the *is_recommended* column in our reviews dataset and performed a series of checks to evaluate its reliability. As per the following:
"""

# Count the number of 1s, 0s, and NAs in the 'is_recommended' column for 3-5 stars reviews
is_recommended_counts = combined_reviews_df['is_recommended'].value_counts(dropna=False)
is_recommended_counts

"""We can already tell that there are a lot of missing values in the *is_recommended* column. Either way, before proceeding, we'll check if any 5-star reviews are incorrectly marked as not recommended, as we would expect these to generally be recommended. This will help ensure data quality."""

# Filter 5-star reviews
five_star_reviews = combined_reviews_df[combined_reviews_df['rating'] == 5]

# Check how many of the 5-star reviews are not recommended
not_recommended_5_stars = five_star_reviews[five_star_reviews['is_recommended'] == 0]

# Count the number of 5-star reviews that are not recommended
not_recommended_count = not_recommended_5_stars.shape[0]

print(f"Number of 5-star reviews marked as not recommended: {not_recommended_count}")

# Show a random review_text from the not recommended 5-star reviews, random seed set for reproducibility
random_example = not_recommended_5_stars['review_text'].sample(1, random_state=42).iloc[0]

print("\nRandom example of 5-star review marked as not recommended:")
print(random_example)

"""Since there are 610 5-star reviews marked as "not recommended," and the random review test shows a very positive review from a happy user, this *is_recommended* column can't be trusted. Therefore, we'll rely on sentiment analysis to accurately identify products users truly recommend and feel good about.

# 3. Sentiment Analysis on Product Reviews

## xlnet-base-cased-product-review-sentiment-analysis

Below we use [xlnet-base-cased-product-review-sentiment-analysis](https://huggingface.co/dipawidia/xlnet-base-cased-product-review-sentiment-analysis) model from Hugging face to perform sentiment analysis.
"""

# Initialize the tokenizer and model
tokenizer = XLNetTokenizer.from_pretrained("dipawidia/xlnet-base-cased-product-review-sentiment-analysis")
model = TFXLNetForSequenceClassification.from_pretrained("dipawidia/xlnet-base-cased-product-review-sentiment-analysis")

# Function to analyze sentiment for a batch of reviews
def get_sentiment_batch(reviews_batch):
    tokenize_text = tokenizer(reviews_batch.tolist(), padding=True, truncation=True, return_tensors='tf', max_length=512)
    preds = model.predict(dict(tokenize_text))['logits']
    class_preds = np.argmax(tf.keras.layers.Softmax()(preds), axis=-1)
    labels = ['Positive' if pred == 1 else 'Negative' for pred in class_preds]
    return labels

# Batch size for processing reviews
batch_size = 64

# Initialize the progress bar for batching
num_batches = len(three_stars) // batch_size + (1 if len(three_stars) % batch_size != 0 else 0)

# List to hold sentiment results
sentiment_results = []

# Process reviews in batches
for i in tqdm(range(0, len(three_stars), batch_size), total=num_batches, desc="Processing reviews", unit="batch"):
    batch = three_stars['review_text'][i:i + batch_size]  # Get the current batch of reviews
    batch_sentiments = get_sentiment_batch(batch)  # Get sentiments for the current batch
    sentiment_results.extend(batch_sentiments)  # Append results

# Assign the sentiment results back to the dataframe
three_stars['sentiment'] = sentiment_results

# Count positive and negative sentiments
sentiment_counts = three_stars['sentiment'].value_counts()
print(sentiment_counts)

# Filter the reviews with positive sentiment
```

```python
positive_reviews_xlnet = three_stars[three_stars['sentiment'] == 'Positive']

# Display the count of positive sentiment reviews
print(f"Number of positive sentiment reviews: {len(positive_reviews_xlnet)}")

print("Positive Sentiment Examples:")
print(positive_reviews_xlnet[['review_text']].head())

"""----

## cardiffnlp/twitter-xlm-roberta-base-sentiment

Below we use [cardiffnlp/twitter-xlm-roberta-base-sentiment](https://huggingface.co/cardiffnlp/twitter-xlm-roberta-base-sentiment) model from Hugging face to perform again sentiment analysis.
"""

# Model path for the new model
model_path = "cardiffnlp/twitter-xlm-roberta-base-sentiment"

# Initialize the tokenizer and model
tokenizer = AutoTokenizer.from_pretrained(model_path)
model = TFAutoModelForSequenceClassification.from_pretrained(model_path)

# Function to analyze sentiment for a batch of reviews
def get_sentiment_batch(reviews_batch):
    # Tokenize the batch of reviews
    tokenize_text = tokenizer(reviews_batch.tolist(), padding=True, truncation=True, return_tensors='tf', max_length=512)

    # Get predictions from the model
    preds = model(**tokenize_text)['logits']

    # Apply softmax to get class probabilities
    scores = softmax(preds.numpy(), axis=-1)

    # Get the class with the highest probability (0 = negative, 2 = positive)
    class_preds = np.argmax(scores, axis=-1)

    # Assign sentiment labels based on the class prediction
    labels = ['Negative' if pred == 0 else 'Neutral' if pred == 1 else 'Positive' for pred in class_preds]

    return labels

# Batch size for processing reviews
batch_size = 64

# Ensure the reviews are valid strings and filter out NaN values
three_stars['review_text'] = three_stars['review_text'].fillna('').astype(str)

# Initialize the progress bar for batching
num_batches = len(three_stars) // batch_size + (1 if len(three_stars) % batch_size != 0 else 0)

# List to hold sentiment results
sentiment_results = []

# Process reviews in batches
for i in tqdm(range(0, len(three_stars), batch_size), total=num_batches, desc="Processing reviews", unit="batch"):
    batch = three_stars['review_text'][i:i + batch_size]  # Get the current batch of reviews
    batch_sentiments = get_sentiment_batch(batch)  # Get sentiments for the current batch
    sentiment_results.extend(batch_sentiments)  # Append results

# Assign the sentiment results back to the dataframe
three_stars['sentiment'] = sentiment_results

# Count positive, negative, and neutral sentiments
sentiment_counts = three_stars['sentiment'].value_counts()
print(sentiment_counts)

# Filter the positive sentiment reviews
positive_reviews_cardiffnlp = three_stars[three_stars['sentiment'] == 'Positive']

# Display the count of positive sentiment reviews
print(f"Number of positive sentiment reviews: {len(positive_reviews_cardiffnlp)}")

print("Positive Sentiment Examples:")
print(positive_reviews_cardiffnlp[['review_text', 'product_id']].head())

"""----

## Intersect all positive reviews found
"""

# Merge the two positive review datasets based on the 'Unnamed: 0' column
intersected_positive_reviews = pd.merge(
```

```
    positive_reviews_xlnet[['Unnamed: 0', 'review_text', 'product_id', 'author_id', 'review_title',
                                'product_name', 'brand_name', 'price_usd']],
    positive_reviews_cardiffnlp[['Unnamed: 0', 'product_id', 'author_id', 'review_title',
                                     'product_name', 'brand_name', 'price_usd']],
    on='Unnamed: 0',
    how='inner'  # 'inner' ensures only matching rows are returned
)

# Rename columns to avoid conflicts (the '_x' and '_y' suffixes)
intersected_positive_reviews.rename(columns={
    'review_text_x': 'review_text',
    'product_id_x': 'product_id',
    'author_id_x': 'author_id',
    'review_title_x': 'review_title',
    'product_name_x': 'product_name',
    'brand_name_x': 'brand_name',
    'price_usd_x': 'price_usd'
}, inplace=True)

# Now drop the columns from the second dataframe that are not needed (with the '_y' suffix)
intersected_positive_reviews.drop(columns=[col for col in intersected_positive_reviews.columns if col.endswith('_y')], inplace=True)

# Select only desired columns
final_columns = ['Unnamed: 0', 'author_id', 'review_text', 'review_title', 'product_id', 'product_name', 'brand_name', 'price_usd']
intersected_positive_reviews = intersected_positive_reviews[final_columns]

# Display the count of intersected positive sentiment reviews
print(f"Number of intersected positive sentiment reviews: {len(intersected_positive_reviews)}")

# Check the first few rows of the final dataframe
print(intersected_positive_reviews.head())

# Save the intersected positive reviews to a CSV file
intersected_positive_reviews.to_csv('processed_data/3_star_positive_reviews.csv', index=False)

"""_____

## Filtering and Combining Desired Reviews
"""

# Load the combined reviews data from the saved file
combined_reviews_df = pd.read_csv('processed_data/combined_reviews.csv', encoding='utf-8', engine='python')

# Filter the reviews with 4 and 5 stars
frequent_items_reviews = combined_reviews_df[combined_reviews_df['rating'].isin([4, 5])]

# Load the positive reviews from the sentiment analysis on the 3-star reviews
positive_reviews_df = pd.read_csv('processed_data/3_star_positive_reviews.csv', encoding='utf-8', engine='python')

# Combine the filtered 4 and 5-star reviews with the 3-star positive reviews
frequent_items_reviews = pd.concat([frequent_items_reviews, positive_reviews_df], ignore_index=True)

print(f"Total reviews to be used in frequent items: {len(frequent_items_reviews)}")

# Check how many unique products there are based on 'product_id'
unique_products_count = frequent_items_reviews['product_id'].nunique()

# Pretty printing the result
print(f"Total number of unique products in the reviews dataset: {unique_products_count}")

"""# 4. Frequent Items"""

# Step 1: Aggregate reviews by user
user_review_counts = frequent_items_reviews.groupby('author_id')['product_id'].nunique()

# Step 2: Define thresholds for grouping
thresholds = list(range(2, 11)) + [20, 30] + list(range(40, user_review_counts.max() + 10, 10))

# Step 3: Count users per group
users_per_group = []

for t in thresholds:
    count = (user_review_counts >= t).sum()
    users_per_group.append(count)

# Step 4: Display results
print("Users per group:")
```

```python
for t, count in zip(thresholds, users_per_group):
    print(f"{t}+ reviews: {count} users")

plt.figure(figsize=(12, 6))
plt.bar([str(t) + "+" for t in thresholds], users_per_group, color="skyblue", alpha=0.8)
plt.title("Number of Users per Group Based on Minimum Reviews")
plt.xlabel("Minimum Reviews per User")
plt.ylabel("Number of Users")
plt.xticks(rotation=45)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

"""## Threshold on users with 10+ reviews"""

# Selecting users with 10+ reviews
filtered_df = frequent_items_reviews.groupby('author_id').filter(lambda x: x['product_id'].nunique() > 9)

# Display the filtered DataFrame
print(f"Number of rows in the filtered dataframe: {len(filtered_df)}")
print(f"Number of unique authors in the filtered dataframe: {len(set(filtered_df['author_id']))}")
#set(filtered_df['author_id'])

# Combining all the products reviewed for each person in the dataset
selected_columns = filtered_df[['author_id', 'product_id']]

# Convert product_id to string before applying 'join'
combined_reviews = selected_columns.groupby('author_id')['product_id'].apply(lambda x: ' '.join(x.astype(str))).reset_index()

print(combined_reviews)

# Convert the 'reviews' into a list of transactions
transactions = combined_reviews['product_id'].str.split().tolist()

# Create a DataFrame for one-hot encoding
# Flatten all unique items (reviews) and create a unique item list
te = TransactionEncoder()
te_array = te.fit(transactions).transform(transactions)
df_encoded = pd.DataFrame(te_array, columns=te.columns_)

# Apply the Apriori algorithm
frequent_itemsets = apriori(df_encoded, min_support=0.03, use_colnames=True)

# Focus on frequent itemsets containing only a single product (not pairs or larger sets)
frequent_pairs = frequent_itemsets[frequent_itemsets['itemsets'].apply(len) == 1]

# Generate association rules, with support as minimum of 3%
rules = association_rules(frequent_itemsets, metric="support", min_threshold=0.03, num_itemsets=len(frequent_itemsets))

print(frequent_pairs)

# Filter rules to include only those where antecedents have a single item
rules_filtered = rules[rules['antecedents'].apply(len) == 1].copy()

rules_display = rules_filtered[["antecedents", "consequents", "support", "confidence", "lift"]].copy()

# Convert frozensets to readable strings
rules_display["antecedents"] = rules_display["antecedents"].apply(lambda x: ', '.join(list(x)))
rules_display["consequents"] = rules_display["consequents"].apply(lambda x: ', '.join(list(x)))

# Sorting by confidence
rules_display = rules_display.sort_values(by="support", ascending=False)
print(rules_display.to_string(index=False))

"""### Testing"""

filtered_rules = rules_display[rules_display['antecedents'].apply(lambda x: 'P500633' in x)]
print(filtered_rules)

# Preprocess the rules DataFrame to ensure 'antecedents' and 'consequents' are sets
def preprocess_rules(rules):
   """
   Preprocess the rules DataFrame to ensure 'antecedents' and 'consequents' are sets.
   """
    # Create a copy of the DataFrame to avoid modifying the original
   rules = rules.copy()
   rules['antecedents'] = rules['antecedents'].apply(lambda x: x.split(", ") if isinstance(x, str) else x)
```

```
    rules['consequents'] = rules['consequents'].apply(lambda x: x.split(", ") if isinstance(x, str) else x)
    return rules

# Function to find items associated with an input item
def find_associated_items(input_item, rules):
    """
    Find all items associated with an input item based on association rules.

    Parameters:
        input_item (str): The item to find associations for.
        rules (pd.DataFrame): A DataFrame of association rules with columns 'antecedents' and 'consequents'.

    Returns:
        list: A list of items associated with the input item, preserving the order of rules.
    """
    # Sort rules by confidence in descending order
    rules = rules.sort_values(by='confidence', ascending=False).reset_index(drop=True)

    input_item_antecedents = [input_item]

    associated_items = []

    # Iterate through the rules
    for _, rule in rules.iterrows():
        antecedents = rule['antecedents']
        consequents = rule['consequents']

        # Check if the input item is in the antecedents
        if input_item_antecedents == antecedents:
            # Add consequents to the associated items if not already present
            for item in consequents:
                if item not in associated_items:
                    associated_items.append(item)

    return associated_items

rules_df = pd.DataFrame(rules_display)

# Preprocess the rules DataFrame
rules_df = preprocess_rules(rules_df)

# Input item
input_item = 'P500633'

# Find associated items
associated_items = find_associated_items(input_item, rules_df)
print(f"Items associated with '{input_item}': {associated_items}")

rules_display.to_csv('processed_data/association_rules.csv', index=False)

"""____

# 5. Similar Items Based on Ingredients and Highlights

### Calculate similar items using ingredients

To calculate similar items using ingredients we choose to embed ingredients with TF—IDF. This was chosen because TF—IDF will take into account the uniqueness of ingredients. For example water is a frequent ingredient in the list which does not tell us a lot about the product. Where as salicylic acid does not occur as often leading to higher importance. To compute the similarity between products we choose cosine similarity because TF—IDF gives results in the form of embedding.

#### Display ingredients
"""

df = pd.read_csv("processed_data/skincare.csv")
# Select only 'Name' and 'Ingredients' columns
df_selected = df[['product_name', 'ingredients']]

# Display the first 5 rows
print(df_selected.head())

"""#### Define functions"""

def custom_tokenizer(text):
    return text.split(", ")

def similarity_search_ingredients(df, query):
    """
    Perform a similarity search based on cosine similarity of TF—IDF vectors.

    Parameters:
    — query (str): The input query string.
    — df (pd.DataFrame): A DataFrame containing 'id' and 'ingredients' columns.

```

```
    Returns:
    - results_df (pd.DataFrame): Rows from the original DataFrame sorted by similarity.
    """
    # Initialize TfidfVectorizer with a custom tokenizer (adjust lowercase as needed)
    vectorizer = TfidfVectorizer(tokenizer=custom_tokenizer, lowercase=False)

    # Extract the 'ingredients' column
    ingredients = df['ingredients']

    # Fit and transform the ingredients
    tfidf_matrix = vectorizer.fit_transform(ingredients)

    # Transform the query into the TF-IDF space
    query_tfidf = vectorizer.transform([query])

    # Compute cosine similarity between the query and all documents
    similarities = cosine_similarity(query_tfidf, tfidf_matrix).flatten()

    # Add similarity scores to the DataFrame
    df['similarity_score_ingredients'] = similarities

    # Sort the DataFrame by similarity scores in descending order
    results_df = df.sort_values(by='similarity_score_ingredients', ascending=False).
        reset_index(drop=True)

    results_df['rank_ingredients'] = range(1, len(results_df) + 1)

    return results_df
"""### Calculate similarity using the highlights

In the dataset we choose we were not provided with description but rather highlights of a
    product. Those are sets of words that describe the product, so therefore there was no
    need to use the minhashing as the length of sets is not big to begging with. To compare
    the sets we choose to use Jaccard Similarity because the more highlight product have in
    common the more similar they are.

#### Display a few rows of highlights
"""

df_selected = df[['product_name', 'highlights']]

# Display the first 5 rows
print(df_selected.head())

"""#### Define functions"""

def jaccard_similarity(set_a, set_b):
    """
    Calculate the Jaccard Similarity between two sets.
    """
    intersection = len(set_a.intersection(set_b))
    union = len(set_a.union(set_b))
    return intersection / union if union != 0 else 0.0

def similarity_search_highlights(df, token_list):
    """
    Perform similarity search based on Jaccard similarity between df and a token list.

    :param df: A pandas DataFrame with columns ['product_id','product_name', 'highlights'].
    :param token_list: A list of tokens to compare against (highlights).
    :return: A DataFrame with IDs and their Jaccard similarity scores, sorted by similarity
        score.
    """
    # Convert the token list to a set
    token_set = set(token_list)

    # List to store the similarity scores
    similarity_scores = []

    # Iterate over the rows of the DataFrame
    for index, row in df.iterrows():
        product_id = row['product_id']
        title = row['product_name']
        # Convert the tokens for this ID to a set
        id_token_set = set(row['highlights'].split(", "))

        # Calculate Jaccard similarity
        similarity_score = jaccard_similarity(id_token_set, token_set)

        # Append the result as a tuple (id, score)
        similarity_scores.append((product_id, title, similarity_score, (row['highlights'])))

    # Convert the list of similarity scores to a DataFrame
    similarity_df = pd.DataFrame(similarity_scores, columns=['product_id', 'product_name', '
        similarity_score_highlights', 'highlights'])

    # Sort the DataFrame by the 'similarity_score' column in descending order
```

```python
    similarity_df_sorted = similarity_df.sort_values(by='similarity_score_highlights', ascending=False).reset_index(drop=True)

    similarity_df_sorted['rank_highlights'] = range(1, len(similarity_df_sorted) + 1)

    return similarity_df_sorted

"""### Combining similarities based on highlight and ingredients
Jaccard similarity measure is in range [0, 1] and cosine similarity is in range [-1, 1] which means that we can not simply calculate the average or weighted sum. Due to that reason we choose to use reciprocal rank fusiona algorithm. To use it we need to rank our similarities tables which is already done in their respective functions.
The **Reciprocal Rank Fusion (RRF)** score for a product _d_ is calculated as:
$$
RRF(d) = \sum_{i=1}^{N} \frac{1}{r_i(d) + k}
\text{}
$$
Where:
- $r_i (d)$ is the rank of product $d$ in the $i^{th}$ ranked list.
- $k$ is a constant (typically $k = 60$), used to prevent division by zero and to adjust the impact of higher ranks.
- $N$ is the number of ranked lists (models or sources).

The reciprocal rank is typically defined as $\frac{1}{r_i(d)}$, where higher ranks give more weight to the document.
"""

def reciprocal_rank_fusion(df_highlights, df_ingredients, k=60):
    """
    Compute Reciprocal Rank Fusion (RRF) scores based on rank_highlights and rank_ingredients.

    Parameters:
    - df_highlights (pd.DataFrame): DataFrame containing 'product_id', 'rank_highlights', and other relevant columns.
    - df_ingredients (pd.DataFrame): DataFrame containing 'product_id', 'rank_ingredients', and other relevant columns.
    - k (int): A constant for RRF computation (default=60).

    Returns:
    - combined_df (pd.DataFrame): A new DataFrame with overall RRF scores and combined ranking.
    """
    # Merge the two DataFrames on 'product_id'
    merged_df = pd.merge(
        df_highlights,  # Include all columns from df_highlights
        df_ingredients[['product_id', 'rank_ingredients']],  # Include only product_id and rank_ingredients
        on='product_id',
        how='inner'
    )

    # Fill missing ranks with a large value (e.g., very low relevance)
    merged_df['rank_highlights'] = merged_df['rank_highlights'].fillna(float('inf'))
    merged_df['rank_ingredients'] = merged_df['rank_ingredients'].fillna(float('inf'))

    # Compute the RRF score
    merged_df['rrf_score'] = (
        1 / (k + merged_df['rank_highlights']) +
        1 / (k + merged_df['rank_ingredients'])
    )

    # Sort by the RRF score in descending order
    merged_df = merged_df.sort_values(by='rrf_score', ascending=False).reset_index(drop=True)

    # Add a new rank based on the RRF score
    merged_df['overall_rank'] = range(1, len(merged_df) + 1)

    return merged_df

"""### Run code for specific product_id"""

def get_similar_items(product_id, df, n = 5):
    """
    Retrieve the top N products most similar to a given product based on highlights and ingredients.

    This function takes a product ID, performs similarity searches on the product's highlights and ingredients,
    and combines the results using a reciprocal rank fusion algorithm. It returns the top N most similar products.

    Parameters:
    ----------
    product_id : int or str
       The ID of the product for which similar items are being searched.
```

```
    n : int, optional
        The number of similar products to return. Default is 5.

    Returns:
    - merged_results (pd.DataFrame): A new DataFrame containing top n similar items
    """

    # get the selected product
    product = df[df['product_id'] == product_id]

    # get the product highlights and ingredients
    product_highlights =  list(product['highlights'])[0].split(", ")
    product_ingredients = str(product['ingredients'])

    # remove the product I am searching for
    df = df[(df['product_id'] != product_id)]

    # perform similarity searches
    highlights_similarity_results = similarity_search_highlights(df, product_highlights)
    ingredients_similarity_results = similarity_search_ingredients(df, product_ingredients)

    # combine similarity searches with reciprocal rank fusion algorithm
    merged_results = reciprocal_rank_fusion(highlights_similarity_results, ingredients_similarity_results)

    # Return only the top-n products
    return merged_results[:n]

df = pd.read_csv("processed_data/skincare.csv")
product_id = "P442001"
product_name = df[df['product_id'] == product_id]['product_name'].iloc[0]
merged = get_similar_items(product_id, df)['product_name'].head(5)

print(f"The most similar products to the {product_name} are: ")
print(merged)

"""_____

# 6.Recommender

To handle cases where a product lacks association rules such as newly introduced items the system calculates
similarity scores against all other products in the database. It identifies the most similar product and
utilizes its association rules to generate recommendations. This approach maintains consistency in the type
of recommendations provided. Similarly, if there are insufficient association rules, the system leverages the
"similar items" method by using related products as a basis to ensure a consistent and diverse output.
"""

# Recommender function
def recommender(product_id, rules, df, n=5):
    """
    Recommend products based on association rules and similarity.

    Parameters:
        product_id (str): The product ID for which recommendations are needed.
        rules (pd.DataFrame): A DataFrame of association rules with columns 'antecedents' and 'consequents'.
        n (int): Number of recommendations to return.

    Returns:
        list: Recommended products.
    """

    # Step 1: Find associated items
    associated_items = find_associated_items(product_id, rules)

    # no rule for chosen item
    if len(associated_items) == 0:
        # find the most similar antecedents to the product_id

        # get all the single antecedents in the rules
        antecedents = [
            list(item)[0]
            for item in rules["antecedents"]
            if len(item) == 1
        ]

        # filter skincare so it searches for the similar items only among the antecedents and product_id
        antecedents.append(product_id)
        df_antecedents = df[df['product_id'].isin(antecedents)]
```

```
        # the most similar antecedents to a product_id
        the_most_similar_product_id = get_similar_items(product_id, df_antecedents, n = 1)['product_id'].iloc[0]

        associated_items = find_associated_items(the_most_similar_product_id, rules)

    # get the top n associated items
    if len(associated_items) >= n:
        return associated_items[:n]

    # if there are not enough  associated items get the similar items to the product_id as well
    else:
        number_of_similar_items = n - len(associated_items)

        # for each of the consequents find similar items and combine them in one dataframe
        similar_items_dataframes = []
        for i in associated_items:
            # get the top number_of_similar_items
            similar_items = get_similar_items(product_id, df)
            similar_items_dataframes.append(similar_items)

        all_similar_items = pd.concat(similar_items_dataframes, axis=0)

        # order is by similarity_score_highlights
        all_similar_items = all_similar_items.sort_values(by='rrf_score', ascending=False).reset_index(drop=True)

        # take only the top number_of_similar_items
        similar_products = list(all_similar_items['product_id'])[:number_of_similar_items]

        return associated_items + similar_products

"""### Load data"""

association_rules = pd.read_csv('processed_data/association_rules.csv')
association_rules = preprocess_rules(association_rules)
skincare_df = pd.read_csv("processed_data/skincare.csv")

"""### Example of usage"""

random_product_id = skincare_df['product_id'].sample(n=1).iloc[0]

product_name = skincare_df[skincare_df['product_id'] == random_product_id]['product_name'].iloc[0]
merged = get_similar_items(random_product_id, skincare_df)['product_name'].head(5)

print(f"The most similar products to the {product_name} are: ")
print(merged)
```

References

[1] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets*. USA: Cambridge University Press, 2nd ed., 2014.

[2] Kaggle, "Sephora products and skincare reviews," 2023. https://www.kaggle.com/datasets/nadyinky/sephora-products-and-skincare-reviews.

[3] "Github repository for the project." https://github.com/raquelmdtum/CTDS-Final-Project.

[4] H. Face, "dipawidia/xlnet-base-cased-product-review-sentiment-analysis." https://huggingface.co/dipawidia/xlnet-base-cased-product-review-sentiment-analysis.

[5] H. Face, "twitter-xlm-roberta-base for sentiment analysis." https://huggingface.co/cardiffnlp/twitter-xlm-roberta-base-sentiment.

[6] D. Shah, "Reciprocal rank fusion (rrf) explained in 4 mins.," 2024.

[7] B. Liu, W. Hsu, and Y. Ma, "Mining association rules with multiple minimum supports," in *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, (New York, NY, USA), p. 337–341, Association for Computing Machinery, 1999.