

Hult SMAFBI Team 3 | A1: Team assignment



Description:

In this assignment, you will work collaboratively to analyze an existing social media campaign and propose enhancements based on your analysis. Your team will use a provided dataset to apply various analytical techniques and generate insights to improve the campaign.

Steps to Complete the Assignment:

Dataset Selection:

Choose a relevant dataset from platforms such as Kaggle from the list:

The COVID-19 Twitter Dataset [Links to an external site.](#), which contains tweets related to COVID-19.

The Amazon Fine Food Reviews [Links to an external site.](#) which contains product and user information, ratings, and a plain text review. It also includes reviews from all other Amazon categories.

The Yelp Dataset [Links to an external site.](#) which contains user reviews, business information, and ratings across various categories.

Ensure the dataset includes text data suitable for sentiment analysis, topic modeling, and other text mining techniques.

Data Collection and Cleaning:

Collaboratively clean and preprocess the dataset to ensure its quality and reliability.

Document the steps taken to clean and prepare the data.

Descriptive Data Analysis and Code Quality:

Explore various attributes of the dataset, such as user demographics, content types, and engagement metrics.

Use Python to generate descriptive statistics and create visualizations that help in understanding the data.

Ensure your code is well-commented, clear, and error-free.

Analytical Techniques:

Apply relevant analytical techniques such as sentiment analysis, topic modeling, and network analysis to uncover patterns and trends within the dataset.

Presentation and Insights:

Prepare a detailed presentation (.ppt or .pdf) summarizing your analysis and proposed enhancements.

Discuss and evaluate the implications of your insights for the chosen business or industry.

Use visual aids, charts, and graphs to effectively communicate your findings.

> Submission format:

Python code (.py or .ipynb).
Dataset file (.xlsx or .csv) if you chose your own one.
Presentation file (.ppt or .pdf).

Great! To get started, here's a suggested breakdown of how you and your team can approach this assignment step by step:

1. Dataset Selection

Choose one of the following based on your team's interest and campaign relevance:

- **COVID-19 Twitter Dataset**
Best for analyzing real-time engagement, public sentiment, and misinformation patterns.
- **Amazon Fine Food Reviews**
Ideal for product feedback analysis and identifying review trends and customer sentiment.
- **Yelp Dataset**
Good for analyzing local business feedback, review patterns, and service performance.

Recommendation:


If your focus is on **enhancing a social media campaign**, the **COVID-19 Twitter Dataset** might offer the richest insights due to real-time tweet engagement data.

2. Data Collection and Cleaning

Use Python libraries like **pandas**, **re**, **nltk**, and **spacy**.

Steps:

- Remove nulls, duplicates
- Tokenize and clean text (remove stopwords, punctuation, etc.)
- Normalize text (lemmatization or stemming)

 Document everything (e.g., in a Jupyter notebook markdown cell).

3. Descriptive Analysis & Code Quality

Explore dataset attributes such as:

- Most active users
- Most frequent words
- Top hashtags, mentions, or product features

Visualizations using **matplotlib**, **plotly**, or **seaborn**:

- Word clouds
- Bar plots of sentiments

- Time series of tweet volumes

Ensure:

- Clear variable names
- Commented functions
- No redundant code

4. Analytical Techniques

Use these techniques:

- **Sentiment Analysis:** VADER or TextBlob
- **Topic Modeling:** LDA or NMF with `gensim` or `sklearn`
- **Network Analysis** (for Twitter): Identify influencers and retweet/reply relationships using `networkx`

Optional:

- Text clustering using **TF-IDF + KMeans**
- NER to identify brands or locations

5. Presentation and Insights

Prepare a 10–15 slide presentation that covers:

1. Introduction and Dataset Overview
2. Cleaning Process
3. Descriptive Statistics
4. Sentiment & Topic Modeling Results
5. Network Analysis (if done)
6. Key Insights
7. Campaign Improvement Suggestions
8. Limitations & Future Work
9. Conclusion

Use Canva, PowerPoint, or Google Slides. Include:

- Clear titles
- Visuals + explanations
- Short recommendations tied to insights

Would you like help picking a dataset or building the notebook structure to begin coding?



give me the outline of the python notebook

Here's a well-structured **Python Notebook Outline** for your social media campaign analysis project. You can use this to organize your `.ipynb` submission:

Social Media Campaign Analysis Notebook Outline

◆ 1. Introduction

markdown

```
# Project Title: Social Media Campaign Analysis
# Team Members: [List names]
# Objective: Analyze [Dataset Name] to uncover insights and propose enhancements for the campaign.
```

◆ 2. Library Imports

python

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import re
import nltk
import spacy
from wordcloud import WordCloud
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import LatentDirichletAllocation
from nltk.sentiment.vader import SentimentIntensityAnalyzer
import networkx as nx
```

◆ 3. Data Loading

python

```
# Load the dataset
df = pd.read_csv('filename.csv') # or .xlsx
df.head()
```

◆ 4. Data Cleaning and Preprocessing

python

```
# Remove nulls, duplicates
df.dropna(inplace=True)
df.drop_duplicates(inplace=True)
```

```
# Text cleaning functions
```



just the outline not the code

Absolutely! Here's the **Python Notebook Outline** (no code, just structure):

Social Media Campaign Analysis – Notebook Outline

1. Introduction

- Project title and description
- Team members
- Dataset chosen and rationale
- Objectives of the analysis

2. Library Imports

- List all necessary Python libraries used

3. Data Loading

- Load dataset
- Brief description of dataset structure (columns, size, types)

4. Data Cleaning and Preprocessing

- Handling missing and duplicate values
- Text normalization (lowercasing, punctuation removal, stopword removal, etc.)
- Tokenization and lemmatization
- Final shape of cleaned data

5. Descriptive Analysis

- Basic statistics (count of posts, users, ratings, etc.)
- Visualization of most frequent terms, hashtags, categories
- Word clouds, bar charts, and time trends

6. Sentiment Analysis

- Tool used (e.g., VADER, TextBlob)
- Sentiment distribution (positive, neutral, negative)
- Visualization of sentiment by category or over time

7. Topic Modeling

- Vectorization (TF-IDF or CountVectorizer)
- LDA or NMF model explanation
- Top words per topic and sample interpretation
- Topic distribution across the dataset

8. Network Analysis (*optional – if Twitter dataset is used*)

- Construction of user or hashtag graph
- Identification of key influencers
- Visualization of network graph

9. Insights and Recommendations

- Key findings from sentiment, topic, and network analysis
- Suggested enhancements for the campaign
- Strategic implications

10. Conclusion

- Summary of methods and insights
- Limitations of analysis
- Future research or application directions

11. References and Appendix

- Cited tools, libraries, papers (if any)
 - Extra plots, code snippets, or alternate analyses
-

Let me know if you'd like a pre-filled template to start working on in Jupyter Notebook or Google Colab.



how can i get Amazon Fine Food Reviews
Analyze ~500,000 food reviews from Amazon from kaggle to google colab

To use the **Amazon Fine Food Reviews dataset** (~500,000 reviews) on **Google Colab**, follow these steps:

✓ Step-by-Step Instructions

♦ Step 1: Get the Dataset from Kaggle

1. Go to the Kaggle dataset page:
<https://www.kaggle.com/datasets/snap/amazon-fine-food-reviews>
2. **Log in** to your Kaggle account.
3. Click "**Download**" OR use your **Kaggle API Token** for direct download in Colab.

♦ Step 2: Set Up Kaggle API in Google Colab

Option A: Upload your Kaggle API token manually

1. In Kaggle:
 - Go to your [Account settings](#).
 - Scroll to **API** section → Click **Create New API Token** → It downloads **kaggle.json**.
2. In Colab:

```
python

from google.colab import files
files.upload() # Upload the kaggle.json file
```

3. Move it and download the dataset:

```
python

!mkdir -p ~/.kaggle
!mv kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json

!kaggle datasets download -d snap/amazon-fine-food-reviews
```

4. Unzip the file:

```
python

!unzip amazon-fine-food-reviews.zip
```

♦ Option B: Upload the CSV manually (if API not available)

1. Download the CSV file **Reviews.csv** to your local machine.
2. In Colab:

```
python

from google.colab import files
uploaded = files.upload()
```

◆ Step 3: Load the Data

Once you have **Reviews.csv** in your Colab workspace:

```
python

import pandas as pd
df = pd.read_csv('Reviews.csv')
df.head()
```

Let me know if you want a ready-made starter Colab notebook with this setup!



```
# Install dependencies as needed:
pip install kagglehub[pandas-datasets]
import kagglehub
from kagglehub import KaggleDatasetAdapter

# Set the path to the file you'd like to load
file_path = ""

# Load the latest version
df = kagglehub.load_dataset(
    KaggleDatasetAdapter.PANDAS,
    "snap/amazon-fine-food-reviews",
    file_path,
    # Provide any additional arguments like
    # sql_query or pandas_kwargs. See the
    # documentation for more information:
    # https://github.com/Kaggle/kagglehub/blob/main/README.md#kaggledatasetadapterpandas
)

print("First 5 records:", df.head())
```

Your code is nearly complete for using the **kagglehub** library to load the **Amazon Fine Food Reviews** dataset into Google Colab. Below is a refined version with the missing pieces filled in:

✅ Final kagglehub Setup to Load Amazon Fine Food Reviews


```
python

# Step 1: Install kagglehub with pandas dataset support
!pip install kagglehub[pandas-datasets]

# Step 2: Import necessary modules
import kagglehub
from kagglehub import KaggleDatasetAdapter

# Step 3: Define the path to the specific CSV file in the dataset
file_path = "Reviews.csv" # This is the main file inside the dataset

# Step 4: Load dataset using kagglehub
df = kagglehub.load_dataset(
    KaggleDatasetAdapter.PANDAS,
    "snap/amazon-fine-food-reviews", # dataset slug from Kaggle
    file_path,
)

# Step 5: Preview the data
print("First 5 records:")
print(df.head())
```

Important Notes:

- The first time you run **kagglehub**, it will ask you to authenticate your Kaggle account — follow the pop-up to connect.
- The **file_path** must match the filename inside the dataset exactly — in this case, it's **"Reviews.csv"**.

Would you like a shareable Colab link that includes this setup ready to go?



all good, we will now proceed with the data cleaning and pre-processing

Perfect! Here's a **structured outline for the Data Cleaning and Preprocessing** section tailored to the **Amazon Fine Food Reviews** dataset:

Data Cleaning & Preprocessing Outline

◆ 1. Initial Inspection

- Check for missing values
- Check for duplicates
- Review column types and key variables (**Score**, **Text**, **Summary**, **Time**, etc.)

◆ 2. Column Selection

Focus on relevant columns:

- **Text** – the full review (main for text mining)

- **Score** – rating (used for sentiment labels if needed)
- **Summary** – short version of review (optional)
- **Time** – timestamp (for time-based trends, optional)

◆ 3. Handling Missing & Duplicate Data

- Remove rows with missing text or score
- Drop duplicates based on **Text** and **UserId** or **ProductId**

◆ 4. Text Preprocessing

Steps to apply on **Text** column:

- Lowercasing
- Remove punctuation, numbers, and special characters
- Remove stopwords
- Tokenization
- Lemmatization (or stemming)

🔧 **Libraries to use:** `nltk`, `spacy`, or `re`

◆ 5. Label Engineering (Optional for Sentiment)

- Convert **Score** to sentiment classes:
 - 1–2: Negative
 - 3: Neutral
 - 4–5: Positive

◆ 6. Save Cleaned Data

- Save preprocessed DataFrame as `cleaned_reviews.csv` or keep it in memory for modeling.

Would you like me to generate the actual code template for these steps now?



we will proceed with the initial inspection and the data cleaning code

Great! Here's the **Python code for Initial Inspection and Data Cleaning** of the Amazon Fine Food Reviews dataset:

1. Initial Inspection

```
python

# View shape and columns
print("Dataset Shape:", df.shape)
print("Columns:", df.columns)

# View data types and missing values
print("\nData Info:")
print(df.info())

# View null values
print("\nMissing Values per Column:")
print(df.isnull().sum())

# Preview sample data
df[['UserId', 'ProductId', 'Score', 'Summary', 'Text']].sample(5)
```

2. Data Cleaning

```
python

# Keep only the necessary columns
df = df[['UserId', 'ProductId', 'Score', 'Summary', 'Text', 'Time']]

# Drop rows with missing text or score
df = df.dropna(subset=['Text', 'Score'])

# Drop duplicate reviews (based on user and text)
df = df.drop_duplicates(subset=['UserId', 'Text'])

# Reset index after cleaning
df.reset_index(drop=True, inplace=True)

# Check cleaned shape
print("Cleaned dataset shape:", df.shape)
```

Let me know when you're ready to proceed with **text preprocessing** (lowercasing, punctuation removal, lemmatization, etc.) or if you'd like to apply a **sentiment label** based on the **Score** column.



check duplicates in the initial inspection

You're right! To complete the **Initial Inspection**, here's the updated code with a **duplicate check** included:

1. Initial Inspection (with duplicate check)

```
python

# Dataset overview
print("Dataset Shape:", df.shape)
print("Column Names:", df.columns.tolist())
```

```
# Data types and non-null counts
print("\nData Info:")
df.info()

# Check missing values
print("\nMissing Values:")
print(df.isnull().sum())

# Check for duplicate rows (entirely identical rows)
total_duplicates = df.duplicated().sum()
print(f"\nTotal Duplicate Rows (exact match): {total_duplicates}")

# Check for duplicate reviews based on UserId and Text
user_text_duplicates = df.duplicated(subset=['UserId', 'Text']).sum()
print(f"Duplicate reviews based on UserId and Text: {user_text_duplicates}")

# Preview sample data
df[['UserId', 'ProductId', 'Score', 'Summary', 'Text']].sample(5, random_state=42)
```

This gives you a complete picture of:

- Nulls
- Exact duplicates
- Likely duplicate reviews (based on the same user submitting the same review)

Let me know if you want to drop both types or keep exact-only.



see duplicates for user id, product id, and text separately

Great idea. Here's how to inspect duplicates **individually for UserId, ProductId, and Text** columns in your dataset:



Check Duplicates Separately by Column

```
python

# Duplicate UserIds – users who submitted more than one review
duplicate_users = df['UserId'].duplicated().sum()
print(f"Duplicate UserIds (multiple reviews by same user): {duplicate_users}")

# Duplicate ProductIds – products with multiple reviews
duplicate_products = df['ProductId'].duplicated().sum()
print(f"Duplicate ProductIds (products reviewed multiple times): {duplicate_products}")

# Duplicate Text – identical review texts (may indicate spam or reused templates)
duplicate_texts = df['Text'].duplicated().sum()
print(f"Duplicate Texts (identical review content): {duplicate_texts}")
```

Optional: View Top Repeaters

To better understand patterns, you can also list:

```
python

# Top users by number of reviews
top_users = df['UserId'].value_counts().head(5)
print("\nTop users by number of reviews:\n", top_users)

# Top products by number of reviews
top_products = df['ProductId'].value_counts().head(5)
print("\nTop reviewed products:\n", top_products)

# Most duplicated review texts (suspicious patterns)
most_common_texts = df['Text'].value_counts().head(5)
print("\nMost repeated review texts:\n", most_common_texts)
```

Let me know if you'd like to drop only the duplicated **Text** entries or if you'd prefer to filter based on suspicious review behavior (e.g., spammy accounts).



preview identical review texts

To preview identical review texts (i.e., the same text used in multiple reviews), use the following code:

Preview Identical Review Texts

```
python

# Find duplicated texts
duplicated_texts = df[df['Text'].duplicated(keep=False)]

# Group by 'Text' and count occurrences
text_groups = duplicated_texts.groupby('Text').size().reset_index(name='Count')

# Filter to see only texts repeated more than once
repeated_texts = text_groups[text_groups['Count'] > 1]

# Merge back to original dataframe to preview full details
repeated_reviews = df[df['Text'].isin(repeated_texts['Text'])]

# Preview the repeated reviews with full info
repeated_reviews = repeated_reviews.sort_values(by='Text')
repeated_reviews[['UserId', 'ProductId', 'Score', 'Summary', 'Text']].head(10)
```

This shows the **actual duplicate text content** along with **UserId**, **ProductId**, and **Score** to help you determine if it's spammy, bot-generated, or just repetitive feedback.

Would you like to filter or remove any of these in the cleaning step?



```
# Initial Inspection
# Dataset overview
print("Dataset Shape:", df.shape)
print("Column Names:", df.columns.tolist())
```

```
# Data types and non-null counts
print("\nData Info:")
df.info()

# Check missing values
print("\nMissing Values:")
print(df.isnull().sum())

# Check for duplicate rows (entirely identical rows)
total_duplicates = df.duplicated().sum()
print(f"\nTotal Duplicate Rows (exact match): {total_duplicates}")

# Check for duplicate reviews based on UserId and Text
user_text_duplicates = df.duplicated(subset=['UserId', 'Text']).sum()
print(f"Duplicate reviews based on UserId and Text: {user_text_duplicates}")

# Duplicate UserIds – users who submitted more than one review
duplicate_users = df['UserId'].duplicated().sum()
print(f"Duplicate UserIds (multiple reviews by same user): {duplicate_users}")

# Duplicate ProductIds – products with multiple reviews
duplicate_products = df['ProductId'].duplicated().sum()
print(f"Duplicate ProductIds (products reviewed multiple times): {duplicate_products}")

# Duplicate Text – identical review texts (may indicate spam or reused templates)
duplicate_texts = df['Text'].duplicated().sum()
print(f"Duplicate Texts (identical review content): {duplicate_texts}")

# Find duplicated texts
duplicated_texts = df[df['Text'].duplicated(keep=False)]

# Group by 'Text' and count occurrences
text_groups = duplicated_texts.groupby('Text').size().reset_index(name='Count')

# Filter to see only texts repeated more than once
repeated_texts = text_groups[text_groups['Count'] > 1]

# Merge back to original dataframe to preview full details
repeated_reviews = df[df['Text'].isin(repeated_texts['Text'])]

# Preview the repeated reviews with full info
repeated_reviews = repeated_reviews.sort_values(by='Text')
repeated_reviews[['UserId', 'ProductId', 'Score', 'Summary', 'Text']].head(10)

# Preview sample data
df[['UserId', 'ProductId', 'Score', 'Summary', 'Text']].sample(5, random_state=42)
```

Your inspection script is **excellent and thorough** — it covers:

- ✓ Dataset shape and column overview
- ✓ Missing values
- ✓ Duplicate checks (exact rows, by **UserId**, **ProductId**, and **Text**)
- ✓ Identification and preview of **repeated review texts**
- ✓ A random sample for quick qualitative review

✓ Output from this code should help you decide:

- Whether to **drop identical review texts** (if they look spammy or templated)
- Whether frequent reviewers (**UserId** duplicates) are **legit or suspicious**
- How **frequently products are reviewed**, useful for weighting or filtering

Let me know when you're ready to move on to the **text preprocessing** phase (lowercasing, stopword removal, lemmatization, etc.) — I can help you structure that step too.



preview duplicate texts

To **preview duplicate review texts** clearly, use this focused snippet that groups and displays repeated review content with how many times each appears:

👁️ Preview Most Common Duplicate Texts

```
python

# Find all duplicated review texts
duplicate_texts_df = df[df['Text'].duplicated(keep=False)]

# Count how many times each text appears
text_counts = duplicate_texts_df['Text'].value_counts().reset_index()
text_counts.columns = ['Text', 'Frequency']

# Preview top repeated review texts
print("Top 10 repeated review texts:\n")
text_counts.head(10)
```

💡 Optional: See which users wrote the same text

```
python

# Merge with original DataFrame to get UserId and ProductId for top repeated texts
top_texts = text_counts.head(5)['Text']
df[df['Text'].isin(top_texts)][['UserId', 'ProductId', 'Score',
'Text']].sort_values('Text').head(10)
```

This will help you:

- Identify copy-pasted or spammy reviews

- Decide whether to remove all duplicated texts or just the top N repeated ones

Let me know if you'd like help with automatic **text deduplication** rules after this.



```
# Duplicate Text – identical review texts (may indicate spam or reused templates)
duplicate_texts = df['Text'].duplicated().sum()
print(f"Duplicate Texts (identical review content): {duplicate_texts}")
```

Preview df

To preview the full **DataFrame** of duplicated review texts (not just the count), here's how you can do it:

✓ Preview Rows with Identical Review Texts

python

```
# Extract all rows where the 'Text' column is duplicated (keep=False returns all occurrences,
not just one)
duplicated_text_df = df[df['Text'].duplicated(keep=False)]

# Sort by text for easier viewing
duplicated_text_df = duplicated_text_df.sort_values(by='Text')

# Preview the first few rows
duplicated_text_df[['UserId', 'ProductId', 'Score', 'Summary', 'Text']].head(10)
```

What this does:

- Shows every review that shares its exact text with at least one other review.
- Includes full details to help you assess whether the duplicates are spammy or legitimate (e.g., same user or different users, same product or not).

Would you like to filter these further, such as keeping only one per group or removing repeated reviews entirely?



count how many

To **count how many review texts are duplicated**, and **how many total rows those duplicates make up**, use the following code:



Count of Duplicated Review Texts

python

```
# Count how many *unique* texts are duplicated
num_unique_duplicated_texts = df['Text'].value_counts()
num_unique_duplicated_texts = num_unique_duplicated_texts[num_unique_duplicated_texts > 1]
print(f"Number of unique duplicated review texts: {len(num_unique_duplicated_texts)}")

# Count how many *rows* have duplicated review texts
duplicated_text_rows = df['Text'].duplicated(keep=False).sum()
print(f"Total number of rows with duplicated review texts: {duplicated_text_rows}")
```

Example Output Explanation:

- **Unique duplicated texts:** 5,000 → means 5,000 different review texts were reused
- **Duplicated text rows:** 20,000 → means those 5,000 texts appeared a total of 20,000 times in the dataset

Let me know if you want to **remove** the duplicates or analyze the **most reused review content** next.



proceed to text preprocessing phase

Great! Here's a structured, modular approach to the **Text Preprocessing Phase** using Python and `nltk` / `spacy` for the **Amazon Fine Food Reviews** dataset.

Text Preprocessing Steps

We'll apply the following in order to the `Text` column:

1. Lowercasing
2. Removing punctuation, numbers, and special characters
3. Removing stopwords
4. Tokenization
5. Lemmatization

Step-by-Step Preprocessing Code

◆ 1. Install and Import Libraries

python

```
# Run this only once
!python -m spacy download en_core_web_sm
import re
import spacy
import nltk
from nltk.corpus import stopwords
```

```
# Download stopwords
nltk.download('stopwords')

# Load SpaCy English model
nlp = spacy.load("en_core_web_sm")

# Define stopwords set
stop_words = set(stopwords.words("english"))
```

◆ 2. Define Cleaning Function

```
python

def preprocess_text(text):
    # Lowercase
    text = text.lower()

    # Remove punctuation and numbers
    text = re.sub(r'^a-z\s', '', text)

    # Tokenize and lemmatize
    doc = nlp(text)
    tokens = [token.lemma_ for token in doc if token.lemma_ not in stop_words and
token.lemma_.strip()]

    # Rejoin tokens into clean string
    return ' '.join(tokens)
```

◆ 3. Apply to Dataset

```
python

# Apply preprocessing to the first 10,000 rows for now (can scale up later)
df['Cleaned_Text'] = df['Text'].iloc[:10000].apply(preprocess_text)

# Preview cleaned data
df[['Text', 'Cleaned_Text']].head(5)
```

Notes:

- Processing all ~500K rows may take **several minutes** — you can scale up gradually (e.g., `iloc[:50000]`).
- The `Cleaned_Text` column will be used for sentiment analysis, topic modeling, etc.

Would you like to now create a **sentiment label** from the **Score** column next?



what's the total number of excel rows for this one

The **Amazon Fine Food Reviews** dataset (`Reviews.csv`) contains approximately:

568,454 rows and **10 columns**

This includes all user reviews across many food-related products. Each row is a single review.

You can confirm the exact number with:

```
python

print("Total rows:", df.shape[0])
```

Let me know if you'd like to:

- Process the full dataset (may take a while), or
- Work on a smaller batch first (e.g., 50,000 reviews) to save time and scale later.



it is taking to long to apply the preprocessing, how can i speed it up

You're absolutely right — applying full **SpaCy lemmatization** across 500K+ reviews can be **very slow**, especially in Google Colab. Here are **6 ways to speed it up**, ranging from simplest to most scalable:

Options to Speed Up Preprocessing

♦ 1. Use `nlp.pipe()` for Faster Batch Processing

SpaCy's `nlp.pipe()` is **much faster** than looping over individual documents.

```
python

def preprocess_texts(texts):
    texts = [re.sub(r'^a-z\s', '', t.lower()) for t in texts] # clean text first
    docs = nlp.pipe(texts, batch_size=1000) # faster batch processing

    cleaned = []
    for doc in docs:
        tokens = [token.lemma_ for token in doc if token.lemma_ not in stop_words and
token.lemma_.strip()]
        cleaned.append(" ".join(tokens))
    return cleaned

# Apply to a subset (start small)
df['Cleaned_Text'] = preprocess_texts(df['Text'].iloc[:50000].tolist())
```

♦ 2. Skip Lemmatization (use regex + stopwords only)

For faster prototyping, remove lemmatization:

```
python

def fast_clean(text):
    text = re.sub(r'^a-z\s', '', text.lower())
    tokens = [word for word in text.split() if word not in stop_words]
    return " ".join(tokens)

df['Cleaned_Text'] = df['Text'].iloc[:100000].apply(fast_clean)
```

▶ Up to **10x faster** than SpaCy, though you lose the benefits of lemmatization.

◆ 3. Use Multi-threading with `swifter` or `pandarallel`

Parallelize `apply()` across cores:

```
bash
```

```
!pip install swifter
```

```
python
```

```
import swifter
df['Cleaned_Text'] = df['Text'].iloc[:100000].swifter.apply(fast_clean)
```

⚠ Works best in local Jupyter; Colab has some threading limits.

◆ 4. Process in Batches and Save Intermediate Files

Avoid doing all 500K at once. Instead:

```
python
```

```
batch = df.iloc[0:50000]
batch['Cleaned_Text'] = preprocess_texts(batch['Text'].tolist())
batch.to_csv('cleaned_batch1.csv', index=False)
```

Then repeat with the next batch.

◆ 5. Use a Pretrained Model for Embeddings Later

If you're using this for modeling, consider skipping full cleaning and using pretrained models like BERT or `tf-idf` + `stopword filtering` for speed.

◆ 6. Run the Full Job on a More Powerful Runtime (e.g., GPU)

- In Colab: `Runtime` > `Change runtime type` > `GPU`
- Use `spaCy` with GPU support (advanced: install `spacy[cuda]`)

Would you like me to optimize your current preprocessing script with `nlp.pipe()` or skip lemmatization for now?



optimize and skip lemmatization

Perfect! Here's an **optimized and faster preprocessing function** that **skips lemmatization** and uses efficient regex + stopwords filtering.

This is ideal for quick turnaround on ~100K+ reviews.

✓ Optimized Text Preprocessing (No Lemmatization)

◆ 1. Setup

```
python

import re
import nltk
from nltk.corpus import stopwords

# Download NLTK stopwords (only once)
nltk.download('stopwords')

# Load English stopwords
stop_words = set(stopwords.words("english"))
```

◆ 2. Fast Preprocessing Function (No SpaCy)

```
python

def fast_clean(text):
    # Lowercase
    text = text.lower()

    # Remove punctuation, numbers, and special characters
    text = re.sub(r'[^a-z\s]', '', text)

    # Tokenize and remove stopwords
    tokens = [word for word in text.split() if word not in stop_words and len(word) > 1]

    # Rejoin cleaned tokens
    return " ".join(tokens)
```

◆ 3. Apply to a Subset (e.g., 100,000 Reviews)

```
python

# Apply to a subset first (adjust number as needed)
df_subset = df.iloc[:100000].copy()
df_subset['Cleaned_Text'] = df_subset['Text'].apply(fast_clean)

# Preview cleaned data
df_subset[['Text', 'Cleaned_Text']].head()
```

✓ Result

- Much faster (can process ~100K rows in a couple of minutes)
- Still removes noise and prepares text for sentiment or topic modeling

Let me know if you'd like help exporting this cleaned data or continuing to **sentiment analysis** or **topic modeling** next!



proceed to feature engineering

Excellent! Now that your data is cleaned, we'll move on to **feature engineering** for text analytics. This step transforms your **Cleaned_Text** into numerical features that can be used for **sentiment analysis**, **topic modeling**, or **clustering**.

Feature Engineering Steps

Common Approaches:

1. **TF-IDF Vectorization** (most useful for topic modeling & clustering)
2. **Count Vectorization** (simpler alternative to TF-IDF)
3. (Optional) **Sentiment Labels** from **Score** for supervised tasks

◆ 1. TF-IDF Vectorization

```
python

from sklearn.feature_extraction.text import TfidfVectorizer

# Limit vocabulary size for performance and remove common & rare words
tfidf = TfidfVectorizer(max_df=0.95, min_df=5, max_features=5000)

# Fit and transform the cleaned text
tfidf_matrix = tfidf.fit_transform(df_subset['Cleaned_Text'])

# Check shape: (n_samples, n_features)
print("TF-IDF Matrix Shape:", tfidf_matrix.shape)
```

◆ 2. Create Sentiment Labels (Optional)

Use **Score** to create positive/neutral/negative sentiment classes:

```
python

def map_sentiment(score):
    if score <= 2:
        return "negative"
    elif score == 3:
        return "neutral"
    else:
        return "positive"

df_subset['Sentiment'] = df_subset['Score'].apply(map_sentiment)
df_subset['Sentiment'].value_counts()
```

This is useful if you're planning to:

- Train a **sentiment classifier**
- Compare **topic modeling by sentiment**

◆ 3. Save Features for Modeling

(Optional but recommended)

```
python

import pandas as pd
import pickle

# Save the cleaned DataFrame
df_subset.to_csv('cleaned_reviews.csv', index=False)

# Save the TF-IDF matrix
with open('tfidf_matrix.pkl', 'wb') as f:
    pickle.dump(tfidf_matrix, f)

# Save the vectorizer for decoding later
with open('tfidf_vectorizer.pkl', 'wb') as f:
    pickle.dump(tfidf, f)
```

You're now ready to apply:

- **Sentiment Analysis**
- **Topic Modeling** (LDA)
- **Clustering** (KMeans on TF-IDF)
- **t-SNE or PCA** for visualization

Would you like to proceed with **topic modeling** or **sentiment analysis** next?



proceed to EDA

Great! Let's now proceed with **Exploratory Data Analysis (EDA)** to understand trends, distributions, and patterns in your dataset before modeling.

Here's a structured plan for **EDA on Amazon Fine Food Reviews**:



Exploratory Data Analysis (EDA)

◆ 1. Sentiment Distribution

If you've created a **Sentiment** column from the **Score**:

```
python

import seaborn as sns
import matplotlib.pyplot as plt
```

```
# Plot sentiment class distribution
sns.countplot(data=df_subset, x='Sentiment', order=['positive', 'neutral', 'negative'])
plt.title("Sentiment Distribution")
plt.xlabel("Sentiment")
plt.ylabel("Number of Reviews")
plt.show()
```

◆ 2. Score Distribution

Even without sentiment classes:

```
python

sns.countplot(data=df_subset, x='Score')
plt.title("Review Score Distribution")
plt.xlabel("Score")
plt.ylabel("Number of Reviews")
plt.show()
```

◆ 3. Review Length Analysis

```
python

# Add review length column
df_subset['Review_Length'] = df_subset['Cleaned_Text'].apply(lambda x: len(x.split()))

# Plot review length distribution
sns.histplot(df_subset['Review_Length'], bins=50, kde=True)
plt.title("Distribution of Review Lengths (in words)")
plt.xlabel("Number of Words")
plt.ylabel("Frequency")
plt.show()
```

◆ 4. Most Frequent Words (WordCloud)

```
python

from wordcloud import WordCloud

# Join all cleaned text into a single string
all_words = ' '.join(df_subset['Cleaned_Text'])

# Generate word cloud
wordcloud = WordCloud(width=800, height=400, background_color='white').generate(all_words)

# Display word cloud
plt.figure(figsize=(15, 7))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title("Most Frequent Words in Reviews")
plt.show()
```

◆ 5. Most Reviewed Products or Users (Optional)


```
python

# Top products
top_products = df_subset['ProductId'].value_counts().head(10)
print("Top 10 Most Reviewed Products:\n", top_products)

# Top users
top_users = df_subset['UserId'].value_counts().head(10)
print("Top 10 Most Active Reviewers:\n", top_users)
```

Let me know if you'd like to visualize:

- **Score vs Review Length**
- **Sentiment over Time**
- Or proceed directly to **Topic Modeling or Sentiment Classification**.



proceed
Score vs Review Length

Sentiment over Time

Perfect! Below are the visualizations for:

1. **Review Score vs. Review Length**
2. **Sentiment Trend Over Time**



1. Score vs. Review Length

This helps understand whether higher/lower scores are associated with longer reviews.

```
python

# Make sure Review_Length column exists
df_subset['Review_Length'] = df_subset['Cleaned_Text'].apply(lambda x: len(x.split()))

# Boxplot: Score vs. Review Length
plt.figure(figsize=(10, 6))
sns.boxplot(data=df_subset, x='Score', y='Review_Length')
plt.title("Review Length by Score")
plt.xlabel("Review Score")
plt.ylabel("Number of Words in Review")
plt.show()
```



2. Sentiment Over Time

This helps track sentiment shifts and trends over time (e.g., by year).

- ◆ **Convert Timestamp to Year**

python

```
from datetime import datetime

# Convert Unix timestamp to datetime
df_subset['Review_Year'] = pd.to_datetime(df_subset['Time'], unit='s').dt.year

# Group by year and sentiment
sentiment_by_year = df_subset.groupby(['Review_Year',
'Sentiment']).size().reset_index(name='Count')

# Pivot for plotting
sentiment_pivot = sentiment_by_year.pivot(index='Review_Year', columns='Sentiment',
values='Count').fillna(0)

# Plot line chart
sentiment_pivot.plot(kind='line', figsize=(12, 6), marker='o')
plt.title("Sentiment Trend Over Time")
plt.xlabel("Year")
plt.ylabel("Number of Reviews")
plt.legend(title='Sentiment')
plt.grid(True)
plt.show()
```

Let me know if you want to proceed with:

- **Topic Modeling (LDA)**
 - **Clustering**
 - **Sentiment Classification (Supervised ML)**
- Or save the cleaned/engineered data before modeling.



```
from datetime import datetime

# Convert Unix timestamp to datetime
df_subset['Review_Year'] = pd.to_datetime(df_subset['Time'], unit='s').dt.year

# Group by year and sentiment
sentiment_by_year = df_subset.groupby(['Review_Year',
'Sentiment']).size().reset_index(name='Count')

# Pivot for plotting
sentiment_pivot = sentiment_by_year.pivot(index='Review_Year', columns='Sentiment',
values='Count').fillna(0)

# Plot line chart
sentiment_pivot.plot(kind='line', figsize=(12, 6), marker='o')
plt.title("Sentiment Trend Over Time")
plt.xlabel("Year")
plt.ylabel("Number of Reviews")
plt.legend(title='Sentiment')
plt.grid(True)
plt.show()
```

```
-----
NameError                                Traceback (most recent call last)
/tmp/ipython-input-52-3772579682.py in <cell line: 0>()
      2
      3 # Convert Unix timestamp to datetime
----> 4 df_subset['Review_Year'] = pd.to_datetime(df_subset['Time'], unit='s').dt.year
      5
      6 # Group by year and sentiment

NameError: name 'pd' is not defined
```

The error you're seeing — **NameError: name 'pd' is not defined** — simply means you **haven't imported pandas yet** (or it was cleared in the current runtime session).

✓ Fix: Just import pandas at the top of your code cell

```
python

import pandas as pd
from datetime import datetime

# Convert Unix timestamp to datetime
df_subset['Review_Year'] = pd.to_datetime(df_subset['Time'], unit='s').dt.year

# Group by year and sentiment
sentiment_by_year = df_subset.groupby(['Review_Year',
'Sentiment']).size().reset_index(name='Count')

# Pivot for plotting
sentiment_pivot = sentiment_by_year.pivot(index='Review_Year', columns='Sentiment',
values='Count').fillna(0)

# Plot line chart
sentiment_pivot.plot(kind='line', figsize=(12, 6), marker='o')
plt.title("Sentiment Trend Over Time")
plt.xlabel("Year")
plt.ylabel("Number of Reviews")
plt.legend(title='Sentiment')
plt.grid(True)
plt.show()
```

Let me know if you want to smooth the lines, normalize values, or annotate key peaks for more insight!



```
use plotly
import seaborn as sns
import matplotlib.pyplot as plt

# Plot sentiment class distribution
sns.countplot(data=df_subset, x='Sentiment', order=['positive', 'neutral', 'negative'])
plt.title("Sentiment Distribution")
plt.xlabel("Sentiment")
```

```
plt.ylabel("Number of Reviews")
plt.show()
```

Here's how to **replace your Seaborn bar chart with an interactive Plotly version** for the **Sentiment Distribution**:

✓ Plotly Version: Sentiment Distribution

```
python

import plotly.express as px

# Count sentiment classes
sentiment_counts = df_subset['Sentiment'].value_counts().reindex(['positive', 'neutral', 'negative'])

# Convert to DataFrame for Plotly
sentiment_df = sentiment_counts.reset_index()
sentiment_df.columns = ['Sentiment', 'Count']

# Plot with Plotly
fig = px.bar(
    sentiment_df,
    x='Sentiment',
    y='Count',
    color='Sentiment',
    title='Sentiment Distribution',
    text='Count'
)

fig.update_layout(
    xaxis_title='Sentiment',
    yaxis_title='Number of Reviews',
    template='plotly_white'
)

fig.show()
```

Let me know if you want this as a **horizontal bar**, **percentage-based chart**, or if you'd like to animate other EDA visuals using Plotly!



```
use plotly
sns.countplot(data=df_subset, x='Score')
plt.title("Review Score Distribution")
plt.xlabel("Score")
plt.ylabel("Number of Reviews")
plt.show()
```

Here's the **Plotly version** of your **Review Score Distribution** chart to replace the Seaborn `countplot`:

✓ Plotly Version: Review Score Distribution

```
python

import plotly.express as px

# Count review scores
score_counts = df_subset['Score'].value_counts().sort_index() # Ensure proper order (1-5)

# Convert to DataFrame for Plotly
score_df = score_counts.reset_index()
score_df.columns = ['Score', 'Count']

# Plot with Plotly
fig = px.bar(
    score_df,
    x='Score',
    y='Count',
    text='Count',
    color='Score', # Optional: gives color per score
    title='Review Score Distribution'
)

fig.update_layout(
    xaxis_title='Score',
    yaxis_title='Number of Reviews',
    template='plotly_white'
)

fig.show()
```

Let me know if you'd like to break this down by **year** or **sentiment**, or convert it into a **pie** or **histogram**!



```
use plotly
# Add review length column
df_subset['Review_Length'] = df_subset['Cleaned_Text'].apply(lambda x: len(x.split()))

# Plot review length distribution
sns.histplot(df_subset['Review_Length'], bins=50, kde=True)
plt.title("Distribution of Review Lengths (in words)")
plt.xlabel("Number of Words")
plt.ylabel("Frequency")
plt.show()
```

Here's the **Plotly version** of your **Review Length Distribution** histogram to replace the Seaborn **histplot**:

Plotly Version: Review Length Distribution

```
python

import plotly.express as px

# Ensure 'Review_Length' column exists
df_subset['Review_Length'] = df_subset['Cleaned_Text'].apply(lambda x: len(x.split()))

# Plot with Plotly
fig = px.histogram(
```

```
df_subset,
x='Review_Length',
nbins=50,
title='Distribution of Review Lengths (in words)',
)

fig.update_layout(
    xaxis_title='Number of Words',
    yaxis_title='Frequency',
    template='plotly_white'
)

fig.show()
```

Let me know if you'd like to:

- Compare review length by **score** or **sentiment**
- Add a **density line**
- Apply **log scale** for long tails



```
use plotly
import pandas as pd
from datetime import datetime

# Convert Unix timestamp to datetime
df_subset['Review_Year'] = pd.to_datetime(df_subset['Time'], unit='s').dt.year

# Group by year and sentiment
sentiment_by_year = df_subset.groupby(['Review_Year',
'Sentiment']).size().reset_index(name='Count')

# Pivot for plotting
sentiment_pivot = sentiment_by_year.pivot(index='Review_Year', columns='Sentiment',
values='Count').fillna(0)

# Plot line chart
sentiment_pivot.plot(kind='line', figsize=(12, 6), marker='o')
plt.title("Sentiment Trend Over Time")
plt.xlabel("Year")
plt.ylabel("Number of Reviews")
plt.legend(title='Sentiment')
plt.grid(True)
plt.show()
```

Here's how to convert your **Matplotlib line chart** of **Sentiment Trend Over Time** into an **interactive Plotly version**:



Plotly Version: Sentiment Trend Over Time

python

```
import plotly.express as px
import pandas as pd

# Ensure datetime conversion is done
df_subset['Review_Year'] = pd.to_datetime(df_subset['Time'], unit='s').dt.year

# Group by year and sentiment
sentiment_by_year = df_subset.groupby(['Review_Year',
'Sentiment']).size().reset_index(name='Count')

# Plot using Plotly
fig = px.line(
    sentiment_by_year,
    x='Review_Year',
    y='Count',
    color='Sentiment',
    markers=True,
    title='Sentiment Trend Over Time'
)

fig.update_layout(
    xaxis_title='Year',
    yaxis_title='Number of Reviews',
    template='plotly_white'
)

fig.show()
```

✅ This version gives you:

- **Interactive hover tooltips**
- Clear comparison across **sentiment classes**
- Zoom/pan options by default

Let me know if you'd like to:

- Smooth the lines
- Normalize counts (e.g., by year total)
- Add annotations for peak sentiment periods



Hult_SMAFBI_Team_3__A1_Team_assignment.pdf
PDF

are there other things that can be improved before we proceed to analytical techniques

Based on your team's excellent work so far and the PDF notebook review, here are **refinements you can make before proceeding to analytical techniques** like sentiment modeling, topic modeling, or clustering:

✅ Suggested Improvements Before Analytics

1. Handle Duplicate Texts

- Over **58,000 unique texts** were duplicated, appearing in **232,915 rows**.
- This could **bias sentiment** or **inflate frequent topics**.

 **Recommendation:** Drop duplicate Text entries:

```
python

df_subset = df_subset.drop_duplicates(subset='Text')
```

2. Use/Enhance Helpfulness Features

- You have **HelpfulnessNumerator** and **HelpfulnessDenominator**, which are often underused.

 **Add helpfulness ratio:**

```
python

df_subset['Helpfulness_Ratio'] = df_subset['HelpfulnessNumerator'] /
df_subset['HelpfulnessDenominator'].replace(0, 1)
```

Then:

- Filter out unhelpful reviews
- Compare sentiment of helpful vs. unhelpful reviews

3. Balance Sentiment Classes (for classification)

- Sentiment is **highly imbalanced**:
 - Positive: 443k
 - Negative: 82k
 - Neutral: 42k

 For modeling, consider:

- **Undersampling positives**
- **Oversampling negatives/neutral** (e.g., using **SMOTE** or simple `.sample()`)

4. N-grams in TF-IDF (for richer topics)

- Your TF-IDF vectorizer currently uses unigrams only.

 Try:


```
python

TfidfVectorizer(ngram_range=(1,2), max_df=0.95, min_df=5, max_features=5000)
```

This will help capture phrases like "not good", "highly recommend", etc.

5. Outlier Filtering for Review Length

- Some reviews may be extremely short or long (noise in topic/sentiment analysis)

🔧 Filter extremes:

```
python

df_subset = df_subset[(df_subset['Review_Length'] > 5) & (df_subset['Review_Length'] < 200)]
```

6. Optional: Add POS tagging for advanced use

For tasks like identifying **adjective-heavy reviews** or extracting **noun phrases** before topic modeling.

Would you like me to generate the code to:

- Drop duplicated texts
 - Add helpfulness ratio
 - Balance sentiment classes
- Before diving into LDA or sentiment modeling?



Add helpfulness ratio

Balance sentiment classes

Before diving into LDA or sentiment modeling?

Perfect! Below are the two enhancements you requested before proceeding to **LDA** or **sentiment classification**:

✓ 1. Add Helpfulness Ratio

```
python

# Replace 0 in denominator to avoid division by zero
df_subset['HelpfulnessDenominator'] = df_subset['HelpfulnessDenominator'].replace(0, 1)

# Create Helpfulness Ratio column
df_subset['Helpfulness_Ratio'] = df_subset['HelpfulnessNumerator'] /
df_subset['HelpfulnessDenominator']
```

```
# Optional: Inspect distribution
df_subset['Helpfulness_Ratio'].describe()
```

You can later filter reviews with `Helpfulness_Ratio >= 0.5` if you want to retain more credible feedback only.

✓ 2. Balance Sentiment Classes (Downsample Positive)

◆ View Current Counts:

```
python

df_subset['Sentiment'].value_counts()
```

◆ Downsample Positive Class to Match Negative Class Size

```
python

from sklearn.utils import resample

# Separate by sentiment
df_positive = df_subset[df_subset['Sentiment'] == 'positive']
df_negative = df_subset[df_subset['Sentiment'] == 'negative']
df_neutral = df_subset[df_subset['Sentiment'] == 'neutral']

# Set target size (match smallest class)
target_size = min(len(df_negative), len(df_neutral))

# Downsample positives
df_positive_bal = resample(df_positive,
                           replace=False,
                           n_samples=target_size,
                           random_state=42)

# Downsample others (optional, if you want perfect balance)
df_negative_bal = resample(df_negative, n_samples=target_size, random_state=42)
df_neutral_bal = resample(df_neutral, n_samples=target_size, random_state=42)

# Combine balanced dataset
df_balanced = pd.concat([df_positive_bal, df_negative_bal, df_neutral_bal])

# Shuffle the rows
df_balanced = df_balanced.sample(frac=1, random_state=42).reset_index(drop=True)

# Check new distribution
df_balanced['Sentiment'].value_counts()
```

You're now ready to:

- Run **LDA topic modeling** (e.g., per sentiment)
- Build a **sentiment classifier** using the balanced set

Would you like to start with **topic modeling** or build a **machine learning model for sentiment prediction** next?

Here's exactly where to **insert the Helpfulness Ratio and Sentiment Balancing** code within your existing notebook structure:

📌 Where to Place These in Your Workflow

✓ 1. Add Helpfulness Ratio

📌 **Location:** Right after initial EDA and before feature engineering.

markdown

```
# 🇺🇸 Helpfulness Ratio Feature Engineering
# (Place after basic EDA like sentiment distribution, review length, etc.)
```

python

```
# Add Helpfulness Ratio (Avoid division by zero)
df_subset['Helpfulness_Ratio'] = df_subset['HelpfulnessNumerator'] /
df_subset['HelpfulnessDenominator'].replace(0, 1)
df_subset['Helpfulness_Ratio'] = df_subset['Helpfulness_Ratio'].clip(upper=1.0)
```

➔ Optionally plot or analyze it:

python

```
import plotly.express as px
fig = px.histogram(df_subset, x='Helpfulness_Ratio', nbins=20, title='Helpfulness Ratio
Distribution')
fig.show()
```

✓ 2. Balance Sentiment Classes

📌 **Location:** After you create the **Sentiment** label (which is already in your Feature Engineering section) but **before** topic modeling or ML.

markdown

```
# 🇺🇸 Balance Sentiment Classes Before Modeling
# (Place after you create df_subset['Sentiment'] and TF-IDF vectors)
```

python

```
from sklearn.utils import resample

# Separate sentiment groups
positive_reviews = df_subset[df_subset['Sentiment'] == 'positive']
neutral_reviews = df_subset[df_subset['Sentiment'] == 'neutral']
negative_reviews = df_subset[df_subset['Sentiment'] == 'negative']

# Downsample and upsample as needed
positive_downsampled = resample(positive_reviews, replace=False, n_samples=40000,
random_state=42)
neutral_upsampled = resample(neutral_reviews, replace=True, n_samples=40000, random_state=42)
negative_upsampled = resample(negative_reviews, replace=True, n_samples=40000, random_state=42)

# Combine
df_balanced = pd.concat([positive_downsampled, neutral_upsampled, negative_upsampled])
df_balanced = df_balanced.sample(frac=1, random_state=42).reset_index(drop=True)

# Confirm balance
print(df_balanced['Sentiment'].value_counts())
```

💡 **After this**, you should:

- Use `df_balanced` instead of `df_subset` for **LDA**, **classification**, or **word clouds**.
- Optionally: re-run TF-IDF on `df_balanced['Cleaned_Text']` if modeling needs updated vectors.

Would you like help setting up LDA on `df_balanced` next?



Hult_SMAFBI_Team_3__A1_Team_assignment (1).pdf
PDF

Excellent — based on your updated notebook (`Hult_SMAFBI_Team_3__A1_Team_assignment (1).pdf`), you've now successfully:

- ✓ Added the **Helpfulness Ratio**
- ✓ Balanced the **Sentiment Classes (40K each)**
- ✓ Conducted extensive **EDA using Plotly**
- ✓ Cleaned and vectorized **TF-IDF features** from over 500K reviews

🧠 You're Ready for Analytical Techniques!



You can now proceed with:

✓ Option 1: Topic Modeling (LDA)

Discover key themes discussed in positive, neutral, and negative reviews.

Recommended steps:

1. Run **LDA** on `df_balanced['Cleaned_Text']`
2. Visualize topics by sentiment
3. Extract top words per topic

✓ Option 2: Sentiment Classification (Supervised ML)

Train a model to predict sentiment (positive/neutral/negative) using TF-IDF features.

Recommended models:

- Logistic Regression
- Random Forest
- XGBoost (if performance is key)

- Evaluate using accuracy, F1-score, confusion matrix

Would you like to:

- Start with **Topic Modeling (LDA)** now, or
- Build a **Sentiment Classifier**?

Let me know and I'll guide you step by step (with optimized code).



Apply relevant analytical techniques such as sentiment analysis, topic modeling, and network analysis to uncover patterns and trends within the dataset.

provide code for each

Here's the full set of **Python code blocks** for applying the three main analytical techniques: **sentiment analysis**, **topic modeling (LDA)**, and **network analysis** using the Amazon Fine Food Reviews dataset.

1. Sentiment Analysis

(Using VADER on the original text)

```
python

from nltk.sentiment.vader import SentimentIntensityAnalyzer
import nltk
nltk.download('vader_lexicon')

# Initialize VADER
vader = SentimentIntensityAnalyzer()

# Apply VADER to original Text column
df_balanced['VADER_Score'] = df_balanced['Text'].apply(lambda x: vader.polarity_scores(x)
['compound'])

# Map VADER scores to sentiment classes
def vader_sentiment(score):
    if score >= 0.05:
        return 'positive'
    elif score <= -0.05:
        return 'negative'
    else:
        return 'neutral'

df_balanced['VADER_Label'] = df_balanced['VADER_Score'].apply(vader_sentiment)

# Compare with original Sentiment label
df_balanced[['Sentiment', 'VADER_Label']].sample(10)
```

2. Topic Modeling (LDA)

```
python

from sklearn.decomposition import LatentDirichletAllocation
from sklearn.feature_extraction.text import CountVectorizer

# Use CountVectorizer for LDA
count_vectorizer = CountVectorizer(max_df=0.95, min_df=5, max_features=3000)
doc_term_matrix = count_vectorizer.fit_transform(df_balanced['Cleaned_Text'])

# Fit LDA Model
lda_model = LatentDirichletAllocation(n_components=5, random_state=42)
lda_model.fit(doc_term_matrix)

# Display top words per topic
words = count_vectorizer.get_feature_names_out()

for i, topic in enumerate(lda_model.components_):
    print(f"\nTopic {i + 1}:")
    top_words = topic.argsort()[-10:][::-1]
    print(", ".join([words[i] for i in top_words]))
```

3. Network Analysis (Twitter-style using co-occurrence)

This builds a **word co-occurrence network** (if real Twitter mentions aren't available).

```
python

import itertools
import networkx as nx
import matplotlib.pyplot as plt

# Use subset of reviews
sample_texts = df_balanced['Cleaned_Text'].sample(500, random_state=42)

# Build co-occurrence edges
edges = []
for text in sample_texts:
    tokens = list(set(text.split())) # remove duplicates per doc
    for pair in itertools.combinations(tokens, 2):
        edges.append(pair)

# Count edge frequency
from collections import Counter
edge_freq = Counter(edges)

# Build network
G = nx.Graph()
for edge, weight in edge_freq.items():
    if weight >= 5: # keep frequent pairs
        G.add_edge(edge[0], edge[1], weight=weight)

# Draw network
plt.figure(figsize=(12, 12))
pos = nx.spring_layout(G, k=0.5)
nx.draw_networkx_nodes(G, pos, node_size=40)
nx.draw_networkx_edges(G, pos, alpha=0.3)
nx.draw_networkx_labels(G, pos, font_size=8)
plt.title("Word Co-occurrence Network (Sampled Reviews)")
plt.axis('off')
plt.show()
```

Summary of Techniques

Technique	Purpose
VADER Sentiment	Rule-based sentiment detection
LDA Topic Model	Discover hidden topics/themes
Network Graph	Visualize word co-occurrence links

Let me know if you'd like to:

- Visualize the **topic distribution per sentiment**
- Plot a **confusion matrix** for VADER vs. actual labels
- Export insights into your presentation slides



Visualize the topic distribution per sentiment

Plot a confusion matrix for VADER vs. actual labels

Excellent! Let's do both:

1. Visualize Topic Distribution per Sentiment

We'll apply LDA separately to each sentiment class and plot topic-word distributions.

Step 1: LDA Topic Modeling by Sentiment

python

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.feature_extraction.text import CountVectorizer

def lda_by_sentiment(df, sentiment_label, n_topics=4):
    print(f"\nTopics for {sentiment_label.upper()} reviews:")

    subset = df[df['Sentiment'] == sentiment_label]

    # Vectorize
    vectorizer = CountVectorizer(max_df=0.95, min_df=5, max_features=2000)
    dtm = vectorizer.fit_transform(subset['Cleaned_Text'])

    lda = LatentDirichletAllocation(n_components=n_topics, random_state=42)
    lda.fit(dtm)

    words = vectorizer.get_feature_names_out()
    for idx, topic in enumerate(lda.components_):
        top_words = [words[i] for i in topic.argsort()[-10:][::-1]]
        print(f"Topic {idx+1}: {' '.join(top_words)}")

# Run LDA per sentiment
for label in ['positive', 'neutral', 'negative']:
    lda_by_sentiment(df_balanced, label)
```

✓ Step 2: Optional — WordCloud per Topic

To visually show dominant words in each topic (ask if needed).



2. Confusion Matrix for VADER vs. Ground Truth

```
python

from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Confusion matrix
cm = confusion_matrix(df_balanced['Sentiment'], df_balanced['VADER_Label'], labels=['positive', 'neutral', 'negative'])

# Plot
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['positive', 'neutral', 'negative'])
disp.plot(cmap='Blues')
plt.title("Confusion Matrix: VADER vs Human Labels")
plt.show()
```

Interpretation

- **LDA by sentiment** shows how topics differ by tone — e.g., positive reviews may mention "love", "delicious", while negative may show "disappointed", "bad", etc.
- The **confusion matrix** shows where VADER agrees or disagrees with the actual labels (e.g., misclassifying neutral as positive).

Let me know if you want help:

- Calculating **precision/recall/F1**
- Plotting topic proportions or coherence
- Exporting this for your team presentation deck



provide model performance for the 3

To evaluate **model performance** for the 3 analytical techniques—**VADER sentiment analysis**, **LDA topic modeling**, and **network analysis**—here's how to interpret and optionally quantify each:

✓ 1. VADER Sentiment Model Performance

Since you already have true labels (**Sentiment**) and VADER predictions (**VADER_Label**), we can compute:


```
python

from sklearn.metrics import classification_report, accuracy_score

# Classification report
print("🔍 VADER Model Performance:")
print(classification_report(df_balanced['Sentiment'], df_balanced['VADER_Label'], digits=3))

# Accuracy
vader_accuracy = accuracy_score(df_balanced['Sentiment'], df_balanced['VADER_Label'])
print(f"✅ VADER Accuracy: {vader_accuracy:.3f}")
```

What to expect:

- **High precision for positive**
- **Lower recall for neutral and negative**
- Rule-based models often overpredict **positive** due to vocabulary bias

✅ 2. LDA Topic Modeling Performance

LDA is **unsupervised**, so we use **coherence score** or **topic interpretability** instead.

✨ Option 1: Human interpretability

- Look at top words per topic and see if they form a coherent theme.

✨ Option 2: Coherence score (if using gensim)

```
python

from gensim.models import CoherenceModel
from gensim.corpora import Dictionary
from gensim.utils import simple_preprocess

# Preprocess
texts = [simple_preprocess(doc) for doc in df_balanced['Cleaned_Text']]
id2word = Dictionary(texts)
corpus = [id2word.doc2bow(text) for text in texts]

# Train LDA in gensim
from gensim.models.ldamodel import LdaModel
lda_model_gensim = LdaModel(corpus=corpus, id2word=id2word, num_topics=5, random_state=42)

# Coherence score
coherence_model = CoherenceModel(model=lda_model_gensim, texts=texts, dictionary=id2word,
coherence='c_v')
coherence_score = coherence_model.get_coherence()
print(f"✅ LDA Coherence Score: {coherence_score:.3f}")
```

- A score of **0.45–0.60** is decent.
- You can improve it by tuning **num_topics**, using bigrams, or filtering more stopwords.

✅ 3. Network Analysis Performance

Network analysis is **exploratory**, but here's how you can quantify its structure:

```
python

# Basic graph stats
print(f"✅ Number of Nodes: {G.number_of_nodes()}")
print(f"✅ Number of Edges: {G.number_of_edges()}")
print(f"✅ Average Degree: {sum(dict(G.degree()).values()) / G.number_of_nodes():.2f}")

# Centrality (Top words)
top_central = sorted(nx.degree_centrality(G).items(), key=lambda x: -x[1])[:10]
print("\n🔗 Top 10 Most Connected Words (by centrality):")
for word, score in top_central:
    print(f"{word}: {score:.3f}")
```

This helps evaluate:

- Whether your network has strong **central connectors** (like “great”, “love”, “bad”)
- How dense or fragmented the word relationships are

📌 Summary Table

Technique	Metric	Typical Range
VADER Sentiment	Accuracy, F1, Confusion Matrix	Accuracy: ~0.65–0.75
LDA Topics	Coherence Score (c_v)	0.45–0.65 (good)
Network Analysis	Node count, centrality, density	Varies (visual)

Would you like help:

- Comparing VADER to a supervised ML model (like Logistic Regression)?
- Plotting a bar chart of topic weights per sentiment?
- Exporting performance metrics into a formatted table for your report?



provide consolidated code for the model performance

Absolutely! Here's the **consolidated code** to evaluate the **model performance** of the three analytical techniques: **VADER Sentiment Analysis**, **LDA Topic Modeling**, and **Network Analysis** in one place.

✅ 🔄 Consolidated Model Performance Code

```
python

# -----
# 1. VADER Sentiment Analysis
# -----
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix,
ConfusionMatrixDisplay
```

```

print("🔍 VADER Sentiment Analysis Performance:\n")
print(classification_report(df_balanced['Sentiment'], df_balanced['VADER_Label'], digits=3))

vader_accuracy = accuracy_score(df_balanced['Sentiment'], df_balanced['VADER_Label'])
print(f"✅ VADER Accuracy: {vader_accuracy:.3f}")

# Confusion matrix
cm = confusion_matrix(df_balanced['Sentiment'], df_balanced['VADER_Label'], labels=['positive',
'neutral', 'negative'])
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['positive', 'neutral',
'negative'])
disp.plot(cmap='Blues')
plt.title("Confusion Matrix: VADER vs. Human Labels")
plt.show()

# -----
# 2. LDA Topic Modeling Coherence
# -----
from gensim.models import CoherenceModel, LdaModel
from gensim.corpora import Dictionary
from gensim.utils import simple_preprocess

# Prepare tokens
texts = [simple_preprocess(doc) for doc in df_balanced['Cleaned_Text']]
id2word = Dictionary(texts)
corpus = [id2word.doc2bow(text) for text in texts]

# Train LDA model
lda_model = LdaModel(corpus=corpus, id2word=id2word, num_topics=5, random_state=42)

# Compute coherence
coherence_model = CoherenceModel(model=lda_model, texts=texts, dictionary=id2word,
coherence='c_v')
coherence_score = coherence_model.get_coherence()

print(f"\n🔍 LDA Topic Modeling Performance:")
print(f"✅ Coherence Score (c_v): {coherence_score:.3f}")

# Show top 10 words for each topic
for idx, topic in lda_model.show_topics(num_topics=5, num_words=10, formatted=False):
    topic_words = ", ".join([word for word, _ in topic])
    print(f"Topic {idx+1}: {topic_words}")

# -----
# 3. Network Analysis Statistics
# -----
import networkx as nx
import itertools

# Sample reviews to build graph
sample_texts = df_balanced['Cleaned_Text'].sample(500, random_state=42)
edges = []
for text in sample_texts:
    tokens = list(set(text.split()))
    for pair in itertools.combinations(tokens, 2):
        edges.append(pair)

# Count edge frequency
from collections import Counter
edge_freq = Counter(edges)

# Build graph with edges appearing at least 5 times
G = nx.Graph()
for edge, weight in edge_freq.items():
    if weight >= 5:
        G.add_edge(edge[0], edge[1], weight=weight)

# Graph performance
print("\n🔍 Network Analysis:")
print(f"✅ Number of Nodes: {G.number_of_nodes()}")
print(f"✅ Number of Edges: {G.number_of_edges()}")
print(f"✅ Average Degree: {sum(dict(G.degree()).values()) / G.number_of_nodes():.2f}")

```

```
# Central words by degree centrality
top_central = sorted(nx.degree_centrality(G).items(), key=lambda x: -x[1])[:10]
print("\n🔗 Top 10 Most Connected Words (by Centrality):")
for word, score in top_central:
    print(f"{word}: {score:.3f}")
```


Output Summary

This code will:

- Print **accuracy, precision, recall, and confusion matrix** for VADER
- Display **LDA coherence score** and top words per topic
- Compute **network size, connectivity, and most central words**

Would you like this exported into a **.txt**, **.csv**, or **PDF** format for your submission or presentation?



 VADER Sentiment Analysis Performance:

```
-----
KeyError                                Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in get_loc(self, key)
    3804     try:
-> 3805         return self._engine.get_loc(casted_key)
    3806     except KeyError as err:

index.pyx in pandas._libs.index.IndexEngine.get_loc()

index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 'VADER_Label'
```

The above exception was the direct cause of the following exception:

```
KeyError                                Traceback (most recent call last)
2 frames
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in get_loc(self, key)
    3810     ):
    3811         raise InvalidIndexError(key)
-> 3812         raise KeyError(key) from err
    3813     except TypeError:
    3814         # If we have a listlike key, _check_indexing_error will raise
```

```
KeyError: 'VADER_Label'
```

The error `KeyError: 'VADER_Label'` means that the `VADER_Label` column doesn't exist in your `df_balanced` DataFrame — likely because the VADER sentiment prediction step wasn't run yet.

Let's fix that now.

✓ Add VADER Sentiment Columns (Before Running Performance Evaluation)

Run this code **before** evaluating model performance:

```
python

# 1. Install and import VADER
import nltk
nltk.download('vader_lexicon')
from nltk.sentiment.vader import SentimentIntensityAnalyzer

# 2. Initialize VADER
vader = SentimentIntensityAnalyzer()

# 3. Compute compound score
df_balanced['VADER_Score'] = df_balanced['Text'].apply(lambda x: vader.polarity_scores(x)
['compound'])

# 4. Map compound score to sentiment label
def vader_sentiment(score):
    if score >= 0.05:
        return 'positive'
    elif score <= -0.05:
        return 'negative'
    else:
        return 'neutral'

df_balanced['VADER_Label'] = df_balanced['VADER_Score'].apply(vader_sentiment)
```

Now you can safely re-run the **VADER model performance code**:

```
python

from sklearn.metrics import classification_report, accuracy_score

print("🔍 VADER Sentiment Analysis Performance:\n")
print(classification_report(df_balanced['Sentiment'], df_balanced['VADER_Label'], digits=3))

vader_accuracy = accuracy_score(df_balanced['Sentiment'], df_balanced['VADER_Label'])
print(f"✅ VADER Accuracy: {vader_accuracy:.3f}")
```

Let me know if you want to visualize where VADER fails — for example, showing top **false positives** or **misclassifications**.



provide revised model performance

Here's your **revised and complete model performance code**, including the **VADER setup fix** and **evaluation for all three analytical techniques** (Sentiment, Topic Modeling, Network Analysis).

Consolidated Model Performance (Revised)

python

```
# =====
#  1. VADER Sentiment Analysis Performance
# =====
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Setup VADER
nltk.download('vader_lexicon')
vader = SentimentIntensityAnalyzer()

# Apply VADER to original text
df_balanced['VADER_Score'] = df_balanced['Text'].apply(lambda x: vader.polarity_scores(x)['compound'])

# Map scores to sentiment labels
def vader_sentiment(score):
    if score >= 0.05:
        return 'positive'
    elif score <= -0.05:
        return 'negative'
    else:
        return 'neutral'

df_balanced['VADER_Label'] = df_balanced['VADER_Score'].apply(vader_sentiment)

# Classification report
print("\n VADER Sentiment Analysis Performance:\n")
print(classification_report(df_balanced['Sentiment'], df_balanced['VADER_Label'], digits=3))

# Accuracy
vader_accuracy = accuracy_score(df_balanced['Sentiment'], df_balanced['VADER_Label'])
print(f"\n VADER Accuracy: {vader_accuracy:.3f}")

# Confusion matrix
cm = confusion_matrix(df_balanced['Sentiment'], df_balanced['VADER_Label'], labels=['positive', 'neutral', 'negative'])
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['positive', 'neutral', 'negative'])
disp.plot(cmap='Blues')
plt.title("Confusion Matrix: VADER vs Human Labels")
plt.show()
```

python

```
# =====
#  2. LDA Topic Modeling Performance
# =====
from gensim.models import CoherenceModel, LdaModel
from gensim.corpora import Dictionary
from gensim.utils import simple_preprocess
```

```
# Prepare tokens for LDA
texts = [simple_preprocess(doc) for doc in df_balanced['Cleaned_Text']]
id2word = Dictionary(texts)
corpus = [id2word.doc2bow(text) for text in texts]

# Train LDA model
lda_model = LdaModel(corpus=corpus, id2word=id2word, num_topics=5, random_state=42)

# Compute coherence
coherence_model = CoherenceModel(model=lda_model, texts=texts, dictionary=id2word,
coherence='c_v')
coherence_score = coherence_model.get_coherence()

print(f"\n📊 LDA Topic Modeling Performance:")
print(f"✅ Coherence Score (c_v): {coherence_score:.3f}")

# Show top words per topic
for idx, topic in lda_model.show_topics(num_topics=5, num_words=10, formatted=False):
    words = [w for w, _ in topic]
    print(f"Topic {idx + 1}: {' '.join(words)}")
```

python

```
# =====
# ✅ 3. Network Analysis Performance
# =====

import networkx as nx
import itertools
from collections import Counter

# Sample reviews and build co-occurrence network
sample_texts = df_balanced['Cleaned_Text'].sample(500, random_state=42)
edges = []
for text in sample_texts:
    tokens = list(set(text.split()))
    for pair in itertools.combinations(tokens, 2):
        edges.append(pair)

# Count edge frequencies
edge_freq = Counter(edges)

# Create graph
G = nx.Graph()
for edge, weight in edge_freq.items():
    if weight >= 5:
        G.add_edge(edge[0], edge[1], weight=weight)

# Graph statistics
print("\n📊 Network Analysis Performance:")
print(f"✅ Number of Nodes: {G.number_of_nodes()}")
print(f"✅ Number of Edges: {G.number_of_edges()}")
print(f"✅ Average Degree: {sum(dict(G.degree()).values()) / G.number_of_nodes():.2f}")

# Top connected words
top_central = sorted(nx.degree_centrality(G).items(), key=lambda x: -x[1])[:10]
print("\n🔗 Top 10 Most Connected Words (by Centrality):")
for word, score in top_central:
    print(f"{word}: {score:.3f}")
```

Let me know if you'd like to:

- Save this into a single `.py` or `.ipynb` file
- Plot LDA **topic proportions per sentiment**
- Compare VADER against a **supervised classifier** like Logistic Regression or XGBoost



```
from sklearn.utils import resample

# Separate sentiment groups
positive_reviews = df_subset[df_subset['Sentiment'] == 'positive']
neutral_reviews = df_subset[df_subset['Sentiment'] == 'neutral']
negative_reviews = df_subset[df_subset['Sentiment'] == 'negative']

# Downsample and upsample as needed
positive_downsampled = resample(positive_reviews, replace=False, n_samples=40000,
                                random_state=42)
neutral_upsampled = resample(neutral_reviews, replace=True, n_samples=40000,
                              random_state=42)
negative_upsampled = resample(negative_reviews, replace=True, n_samples=40000,
                               random_state=42)

# Combine
df_balanced = pd.concat([positive_downsampled, neutral_upsampled, negative_upsampled])
df_balanced = df_balanced.sample(frac=1, random_state=42).reset_index(drop=True)

# Confirm balance
print(df_balanced['Sentiment'].value_counts())

-----
NameError                                Traceback (most recent call last)
/tmp/ipython-input-13-2653519299.py in <cell line: 0>()
     12
     13 # Combine
--> 14 df_balanced = pd.concat([positive_downsampled, neutral_upsampled, negative_upsampled])
     15 df_balanced = df_balanced.sample(frac=1, random_state=42).reset_index(drop=True)
     16

NameError: name 'pd' is not defined
```

The error **NameError: name 'pd' is not defined** means that you haven't imported the **pandas** library yet in the current session.

✓ Fix: Import pandas before using it

Add this line **at the top** of your code:

```
python

import pandas as pd
```

🔄 Full Corrected Code Block:

python

```
import pandas as pd
from sklearn.utils import resample

# Separate sentiment groups
positive_reviews = df_subset[df_subset['Sentiment'] == 'positive']
neutral_reviews = df_subset[df_subset['Sentiment'] == 'neutral']
negative_reviews = df_subset[df_subset['Sentiment'] == 'negative']

# Downsample and upsample as needed
positive_downsampled = resample(positive_reviews, replace=False, n_samples=40000,
                                random_state=42)
neutral_upsampled = resample(neutral_reviews, replace=True, n_samples=40000, random_state=42)
negative_upsampled = resample(negative_reviews, replace=True, n_samples=40000, random_state=42)

# Combine and shuffle
df_balanced = pd.concat([positive_downsampled, neutral_upsampled, negative_upsampled])
df_balanced = df_balanced.sample(frac=1, random_state=42).reset_index(drop=True)

# Confirm balance
print(df_balanced['Sentiment'].value_counts())
```

Once you've re-run this, you can proceed with modeling or analysis using **df_balanced**. Let me know if you'd like to reapply VADER or TF-IDF on this new balanced DataFrame.



```
from wordcloud import WordCloud

# Join all cleaned text into a single string
all_words = ' '.join(df_subset['Cleaned_Text'])

# Generate word cloud
wordcloud = WordCloud(width=800, height=400, background_color='white').generate(all_words)

# Display word cloud
plt.figure(figsize=(15, 7))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title("Most Frequent Words in Reviews")
plt.show()

NameError                                Traceback (most recent call last)
/tmp/ipython-input-30-3930676694.py in <cell line: 0>()
      8
      9 # Display word cloud
----> 10 plt.figure(figsize=(15, 7))
      11 plt.imshow(wordcloud, interpolation='bilinear')
      12 plt.axis('off')

NameError: name 'plt' is not defined
```

The error **NameError: name 'plt' is not defined** means that **matplotlib.pyplot** hasn't been imported yet.

✓ Fix: Import matplotlib.pyplot as plt

Just add this line before your plotting code:

```
python

import matplotlib.pyplot as plt
```

↻ Full Corrected Word Cloud Code:

```
python

import matplotlib.pyplot as plt
from wordcloud import WordCloud

# Join all cleaned text into a single string
all_words = ' '.join(df_subset['Cleaned_Text'])

# Generate word cloud
wordcloud = WordCloud(width=800, height=400, background_color='white').generate(all_words)

# Display word cloud
plt.figure(figsize=(15, 7))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title("Most Frequent Words in Reviews")
plt.show()
```

Let me know if you'd like to:

- Generate **separate word clouds per sentiment**
- Filter out very common words
- Use **bigram frequency** for richer clouds



```
# Make sure Review_Length column exists
df_subset['Review_Length'] = df_subset['Cleaned_Text'].apply(lambda x: len(x.split()))

# Boxplot: Score vs. Review Length
plt.figure(figsize=(10, 6))
sns.boxplot(data=df_subset, x='Score', y='Review_Length')
plt.title("Review Length by Score")
plt.xlabel("Review Score")
plt.ylabel("Number of Words in Review")
plt.show()
```

```
-----
NameError                                Traceback (most recent call last)
/tmp/ipython-input-17-142424582.py in <cell line: 0>()
      4 # Boxplot: Score vs. Review Length
```

```
5 plt.figure(figsize=(10, 6))
----> 6 sns.boxplot(data=df_subset, x='Score', y='Review_Length')
7 plt.title("Review Length by Score")
8 plt.xlabel("Review Score")
```

NameError: name 'sns' is not defined
<Figure size 1000x600 with 0 Axes>

The error **NameError: name 'sns' is not defined** means that **seaborn has not been imported**.

✓ Fix: Import Seaborn

Add this import line at the top of your notebook or before the plot:

```
python

import seaborn as sns
```

🔄 Full Corrected Boxplot Code:

```
python

import seaborn as sns
import matplotlib.pyplot as plt

# Ensure Review_Length column exists
df_subset['Review_Length'] = df_subset['Cleaned_Text'].apply(lambda x: len(x.split()))

# Boxplot: Score vs. Review Length
plt.figure(figsize=(10, 6))
sns.boxplot(data=df_subset, x='Score', y='Review_Length')
plt.title("Review Length by Score")
plt.xlabel("Review Score")
plt.ylabel("Number of Words in Review")
plt.show()
```

Let me know if you'd like a **Plotly version** of this boxplot or want to split by **Sentiment** instead of **Score**.



why it is important to use feature engineering here

Great question! Feature engineering is **crucial** in a project like yours involving Amazon Fine Food Reviews for several reasons—especially when working with **text data** and applying **analytical techniques** like sentiment analysis, topic modeling, and classification.

🔍 Why Feature Engineering Is Important in This Context

1. Transforms Raw Text into Usable Inputs

- Raw reviews are unstructured strings.
- Feature engineering transforms these into numerical and categorical features like:
 - **Review_Length** (number of words)
 - **Sentiment Score** (e.g., VADER compound score)
 - **TF-IDF vectors** or **Bag-of-Words**
 - **Helpfulness Ratio** (e.g., helpful votes / total votes)

Without this transformation, models like LDA or classifiers can't "understand" the data.

2. Improves Model Accuracy and Interpretability

- Features like **Review_Length**, **Has_Exclamation**, or **Caps_Count** help **identify emotion or exaggeration**, which are key for sentiment classification.
- Engineered features capture **non-obvious signals** the model may miss in raw text.

Example: Longer reviews may correlate with stronger opinions (positive or negative).

3. Enables Better Visualization and EDA

- Engineered features allow for clearer insights in:
 - Boxplots (Score vs. Length)
 - Word clouds (Frequent terms by sentiment)
 - Sentiment over time (using **Review_Year**)

4. Supports Text Mining Techniques

- **Topic modeling (LDA)** and **network analysis** require tokenized, cleaned data.
 - Features like:
 - Lemmatized or stemmed tokens
 - N-grams
 - Stopword removalenable meaningful topics and connections.
-

5. Balances and Clarifies Bias

- You can detect review manipulation or spam using:
 - Duplicate texts
 - Overused templates
 - Abnormal length or repetition patterns
-

✓ Summary

Without Feature Engineering	With Feature Engineering
Models see text as unreadable blobs	Models receive structured numerical input
High noise, low signal	Distilled insights from review text
Hard to visualize or explore	Clear patterns via length, score, sentiment, etc.
Unbalanced insights	Features like helpfulness or year balance trends

If you like, I can list **specific features** you've already used and suggest **more advanced ones** you can add. Would you like that?



```
# Top products
top_products = df_subset['ProductId'].value_counts().head(10)
print("Top 10 Most Reviewed Products:\n", top_products)

# Top users
top_users = df_subset['UserId'].value_counts().head(10)
print("Top 10 Most Active Reviewers:\n", top_users)

# Group by user ID and profile name, count the number of reviews
top_reviewers = df_clean.groupby(['UserId', 'ProfileName']).size().reset_index(name='ReviewCount')

# Sort the reviewers by number of reviews in descending order
top_reviewers_sorted = top_reviewers.sort_values(by='ReviewCount', ascending=False)

# Select top 10 reviewers
top_10_reviewers = top_reviewers_sorted.head(n=10)

# Plot as a horizontal bar chart
plt.figure(figsize=(10, 6))
sns.barplot(
    x='ReviewCount',
    y='ProfileName',
    data=top_10_reviewers,
    palette='magma'
)

# Add labels and title
plt.title('Top 10 Most Active Reviewers', fontsize=14)
plt.xlabel('Number of Reviews')
plt.ylabel('Reviewer Profile Name')
plt.tight_layout()
plt.show()
```

NameError

Traceback (most recent call last)

```
/tmp/ipython-input-1-3967912387.py in <cell line: 0>()
    1 # Top products
----> 2 top_products = df_subset['ProductId'].value_counts().head(10)
    3 print("Top 10 Most Reviewed Products:\n", top_products)
    4
    5 # Top users
```

NameError: name 'df_subset' is not defined

The error **NameError: name 'df_subset' is not defined** means the variable `df_subset` hasn't been defined in your current runtime — likely because you've restarted the notebook or didn't run earlier cells that define it.

✓ Fix Options

✓ **Option 1: If you're working with `df_balanced` now, simply replace `df_subset` with `df_balanced`:**

```
python

top_products = df_balanced['ProductId'].value_counts().head(10)
print("Top 10 Most Reviewed Products:\n", top_products)

top_users = df_balanced['UserId'].value_counts().head(10)
print("Top 10 Most Active Reviewers:\n", top_users)

# Group by user ID and profile name
top_reviewers = df_balanced.groupby(['UserId',
'ProfileName']).size().reset_index(name='ReviewCount')

# Sort and plot
top_10_reviewers = top_reviewers.sort_values(by='ReviewCount', ascending=False).head(10)

import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
sns.barplot(
    x='ReviewCount',
    y='ProfileName',
    data=top_10_reviewers,
    palette='magma'
)
plt.title('Top 10 Most Active Reviewers', fontsize=14)
plt.xlabel('Number of Reviews')
plt.ylabel('Reviewer Profile Name')
plt.tight_layout()
plt.show()
```

✓ **Option 2: If you're still using `df_subset`, make sure to define it first:**

```
python

# Example: subset of original dataset (define only if needed)
df_subset = df.copy() # or df_clean, df_raw, etc.
```

Let me know if you'd like:

- A **Plotly bar chart** version of the top reviewers
- To extract **top reviewers per product**
- To analyze **potential review spammers**



all

Perfect — here's a complete solution covering **all** your requests:

✓ Part 1: Top Products and Users (using df_balanced)

python

```
# Top 10 most reviewed products
top_products = df_balanced['ProductId'].value_counts().head(10)
print("📊 Top 10 Most Reviewed Products:\n", top_products)

# Top 10 most active reviewers
top_users = df_balanced['UserId'].value_counts().head(10)
print("\n👤 Top 10 Most Active Reviewers:\n", top_users)
```

✓ Part 2: Top 10 Reviewers — Matplotlib & Seaborn Bar Chart

python

```
import matplotlib.pyplot as plt
import seaborn as sns

# Group by UserId and ProfileName
top_reviewers = df_balanced.groupby(['UserId',
'ProfileName']).size().reset_index(name='ReviewCount')
top_10_reviewers = top_reviewers.sort_values(by='ReviewCount', ascending=False).head(10)

# Horizontal bar chart
plt.figure(figsize=(10, 6))
sns.barplot(
    x='ReviewCount',
    y='ProfileName',
    data=top_10_reviewers,
    palette='magma'
)
plt.title('Top 10 Most Active Reviewers', fontsize=14)
plt.xlabel('Number of Reviews')
plt.ylabel('Reviewer Profile Name')
plt.tight_layout()
plt.show()
```

✓ Part 3: 🔄 Plotly Bar Chart (Interactive Version)

```
python

import plotly.express as px

fig = px.bar(
    top_10_reviewers,
    x='ReviewCount',
    y='ProfileName',
    orientation='h',
    title='Top 10 Most Active Reviewers',
    color='ReviewCount',
    color_continuous_scale='Magma'
)
fig.update_layout(yaxis=dict(autorange="reversed"))
fig.show()
```

✓ Part 4: Top Reviewers per Product (Pivoted)

```
python

# Count reviews per (ProductId, UserId)
product_user_counts = df_balanced.groupby(['ProductId',
'UserId']).size().reset_index(name='ReviewCount')

# Get top 3 reviewers per product
top_reviewers_per_product = product_user_counts.sort_values(['ProductId', 'ReviewCount'],
ascending=[True, False])
top_reviewers_per_product = top_reviewers_per_product.groupby('ProductId').head(3)

# Preview
top_reviewers_per_product.head(10)
```

✓ Part 5: Detect Potential Spam Reviewers

These might be users who:

- Post **many reviews** (top percentile)
- Use **duplicate review texts**
- Have unusually short or similar review lengths

```
python

# Identify duplicate reviews by UserId + Text
duplicate_reviewers = df_balanced[df_balanced.duplicated(subset=['UserId', 'Text'], keep=False)]

# Group by user to find frequent duplicators
suspicious_users =
duplicate_reviewers.groupby('UserId').size().reset_index(name='DuplicateReviews')
suspicious_users = suspicious_users.sort_values(by='DuplicateReviews', ascending=False)

# Preview suspicious users
suspicious_users.head(10)
```

Would you like me to export these results (e.g., top reviewer list or suspected spammers) into a downloadable .csv or .xlsx?



```
# Top products
top_products = df_subset['ProductId'].value_counts().head(10)
print("Top 10 Most Reviewed Products:\n", top_products)

# Top users
top_users = df_subset['ProfileName'].value_counts().head(10)
print("Top 10 Most Active Reviewers:\n", top_users)

# Group by user ID and profile name, count the number of reviews
top_reviewers = df_clean.groupby(['UserId', 'ProfileName']).size().reset_index(name='ReviewCount')

# Sort the reviewers by number of reviews in descending order
top_reviewers_sorted = top_reviewers.sort_values(by='ReviewCount', ascending=False)

# Select top 10 reviewers
top_10_reviewers = top_reviewers_sorted.head(n=10)

# Plot as a horizontal bar chart
plt.figure(figsize=(10, 6))
sns.barplot(
    x='ReviewCount',
    y='ProfileName',
    data=top_10_reviewers,
    palette='magma'
)

# Add labels and title
plt.title('Top 10 Most Active Reviewers', fontsize=14)
plt.xlabel('Number of Reviews')
plt.ylabel('Reviewer Profile Name')
plt.tight_layout()
plt.show()

merge, make use of plotly instead
```

Here's a **merged and optimized version** of your code using **Plotly** for interactive visuals. This script includes:

1. Top 10 most reviewed products
2. Top 10 most active reviewers by profile name
3. Interactive horizontal bar chart using Plotly



Cleaned + Merged Code (Plotly Version)

python

```
import pandas as pd
import plotly.express as px

# Top 10 most reviewed products
top_products = df_subset['ProductId'].value_counts().head(10).reset_index()
top_products.columns = ['ProductId', 'ReviewCount']
print("\ud83d\udc1d Top 10 Most Reviewed Products:\n", top_products)

# Top 10 most active reviewers by ProfileName
top_users = df_subset['ProfileName'].value_counts().head(10).reset_index()
top_users.columns = ['ProfileName', 'ReviewCount']
print("\n\ud83d\udc1d Top 10 Most Active Reviewers:\n", top_users)

# Group by UserId and ProfileName to get more accurate top reviewer info
top_reviewers = df_clean.groupby(['UserId', 'ProfileName']).size().reset_index(name='ReviewCount')
top_10_reviewers = top_reviewers.sort_values(by='ReviewCount', ascending=False).head(10)

# Plot using Plotly
fig = px.bar(
    top_10_reviewers,
    x='ReviewCount',
    y='ProfileName',
    orientation='h',
    color='ReviewCount',
    color_continuous_scale='Magma',
    title='Top 10 Most Active Reviewers (ProfileName)'
)

fig.update_layout(
    yaxis=dict(autorange="reversed"),
    xaxis_title='Number of Reviews',
    yaxis_title='Reviewer Profile Name',
    template='plotly_white'
)

fig.show()
```

Notes:

- This version uses **Plotly for better interactivity**.
- It uses both **df_subset** (for top product + profile frequency) and **df_clean** (for **UserId** + **ProfileName** combination).
- **autorange="reversed"** ensures the top reviewer is on top in the horizontal bar chart.

Let me know if you'd like to:

- Include **hover tooltips** with UserID and ReviewCount
- Export results to **.csv**
- Add **top products** as a separate Plotly chart



```
# Group by UserId and ProfileName, and calculate the average score
user_avg_scores = df_clean.groupby(['UserId', 'ProfileName'])['Score'].mean().reset_index()

# Rename column for clarity
```

```

user_avg_scores.rename(columns={'Score': 'AverageScore'}, inplace=True)

# Sort by average score (optional)
user_avg_scores_sorted = user_avg_scores.sort_values(by='AverageScore', ascending=False)

# Display a graph
plt.figure(figsize=(10, 5))
sns.kdeplot(user_avg_scores['AverageScore'], shade=True, color='skyblue', linewidth=2)

# Add labels and title
plt.title("KDE Plot of Average Scores Given per User", fontsize=14)
plt.xlabel("Average Score")
plt.ylabel("Density")
plt.xlim(1, 5)
plt.grid(True)
plt.tight_layout()
plt.show()

use plotly

```

Here's the **Plotly** version of your **KDE plot for average review scores per user** — fully interactive and visually appealing:

Plotly KDE Plot: Average Scores per User

```

python

import pandas as pd
import plotly.figure_factory as ff

# Group by UserId and ProfileName, calculate average score
user_avg_scores = df_clean.groupby(['UserId', 'ProfileName'])['Score'].mean().reset_index()
user_avg_scores.rename(columns={'Score': 'AverageScore'}, inplace=True)

# Extract the data for KDE plot
score_data = user_avg_scores['AverageScore'].values.tolist()

# Create KDE plot using Plotly
fig = ff.create_distplot(
    [score_data], # List of distributions
    group_labels=["Average Scores"], # Legend label
    show_hist=False, # KDE only (no histogram)
    colors=['skyblue']
)

fig.update_layout(
    title="KDE Plot of Average Scores Given per User",
    xaxis_title="Average Score",
    yaxis_title="Density",
    xaxis=dict(range=[1, 5]),
    template='plotly_white'
)

fig.show()

```

Output:

This will display:

- A smooth **density curve** of how generous (or harsh) users are with their scores.
- Interactivity: hover to see precise values.
- No histogram, but you can enable it by changing `show_hist=True`.

Let me know if you'd like to:

- Compare **top vs. bottom reviewers** by score visually
- Overlay **multiple distributions** (e.g., verified vs. unverified reviewers)



```
# Filter to avoid division by zero in helpfulness ratio
df_helpful = df_clean[df_clean['HelpfulnessDenominator'] > 0].copy()

# Compute helpfulness ratio for each review
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator']

# Group by user to get review count and average helpfulness
user_helpfulness = df_helpful.groupby(['UserId', 'ProfileName']).agg(
    ReviewCount=('Text', 'count'),
    AvgHelpfulness=('HelpfulnessRatio', 'mean')
).reset_index()

# Create review count bins
bins = [1, 10, 50, 100, 200, 500, 1000]
labels = ['1-10', '11-50', '51-100', '101-200', '201-500', '500+']
user_helpfulness['ReviewCountBin'] = pd.cut(user_helpfulness['ReviewCount'], bins=bins,
labels=labels, include_lowest=True)

# Plot box plot
plt.figure(figsize=(10, 6))
sns.boxplot(
    x='ReviewCountBin',
    y='AvgHelpfulness',
    data=user_helpfulness,
    palette='coolwarm'
)

# Add labels and title
plt.title('Helpfulness Ratio by Reviewer Activity Level', fontsize=14)
plt.xlabel('Number of Reviews (Binned)')
plt.ylabel('Average Helpfulness Ratio')
plt.tight_layout()
plt.show()
```

Here's the **Plotly** version of your **Helpfulness Ratio by Reviewer Activity Level** box plot — fully interactive and styled for presentation.

Plotly Box Plot: Helpfulness Ratio by Review Count Bin

python

```
import pandas as pd
import plotly.express as px

# Filter to avoid division by zero in helpfulness ratio
df_helpful = df_clean[df_clean['HelpfulnessDenominator'] > 0].copy()

# Compute helpfulness ratio
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator']

# Group by user to calculate average helpfulness and review count
user_helpfulness = df_helpful.groupby(['UserId', 'ProfileName']).agg(
    ReviewCount=('Text', 'count'),
    AvgHelpfulness=('HelpfulnessRatio', 'mean')
).reset_index()

# Create review count bins
bins = [1, 10, 50, 100, 200, 500, 1000]
labels = ['1-10', '11-50', '51-100', '101-200', '201-500', '500+']
user_helpfulness['ReviewCountBin'] = pd.cut(user_helpfulness['ReviewCount'], bins=bins,
labels=labels, include_lowest=True)

# Plot using Plotly
fig = px.box(
    user_helpfulness,
    x='ReviewCountBin',
    y='AvgHelpfulness',
    color='ReviewCountBin',
    title='👉 Helpfulness Ratio by Reviewer Activity Level',
    labels={
        'ReviewCountBin': 'Number of Reviews (Binned)',
        'AvgHelpfulness': 'Average Helpfulness Ratio'
    },
    template='plotly_white'
)

fig.update_layout(showlegend=False)
fig.show()
```

What This Shows:

- How reviewer helpfulness (based on community votes) varies by how **active** a reviewer is.
- The **boxplot** gives insights into **distribution, outliers, and medians** across different bins.

Let me know if you'd like to:

- Add tooltips with exact usernames or review counts
- Export this as an image or include it in your team report



```
# Extract month from timestamp
df_clean['Month'] = df_clean['Time'].dt.month

# Aggregate: Total reviews by month (across all years)
monthly_reviews = df_clean['Month'].value_counts().sort_index()

# Map month numbers to names for readability
month_names = [calendar.month_name[i] for i in monthly_reviews.index]

# Plotting
plt.figure(figsize=(10, 6))
sns.lineplot(x=month_names, y=monthly_reviews.values, marker='o')

# Add labels and title
plt.title("Seasonal Review Patterns: Reviews by Month", fontsize=14)
plt.xlabel("Month")
plt.ylabel("Number of Reviews")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

use plotly
```

Here's the **Plotly version** of your **Seasonal Review Patterns (Reviews by Month)** chart — fully interactive and styled for insight.

Plotly Line Chart: Monthly Review Counts

```
python

import pandas as pd
import plotly.express as px
import calendar

# Ensure 'Time' is in datetime format
df_clean['Time'] = pd.to_datetime(df_clean['Time'], unit='s', errors='coerce')

# Extract month from timestamp
df_clean['Month'] = df_clean['Time'].dt.month

# Aggregate: total reviews per month (across all years)
monthly_reviews = df_clean['Month'].value_counts().sort_index()

# Map numeric months to full month names
month_names = [calendar.month_name[i] for i in monthly_reviews.index]

# Create DataFrame for Plotly
monthly_df = pd.DataFrame({
    'Month': month_names,
    'ReviewCount': monthly_reviews.values
})

# Plot using Plotly
fig = px.line(
    monthly_df,
    x='Month',
    y='ReviewCount',
```

```

markers=True,
title='Seasonal Review Patterns: Reviews by Month',
labels={'Month': 'Month', 'ReviewCount': 'Number of Reviews'},
template='plotly_white'
)

fig.update_traces(line=dict(width=3))
fig.update_layout(xaxis_tickangle=45)
fig.show()

```

✓ Output:

- Line chart showing how reviews vary across months (e.g., holiday spikes).
- Month names are human-readable.
- Fully interactive: hover tooltips, zoom, pan, etc.

Let me know if you'd like to:

- Compare month-wise sentiment patterns
- Normalize this by year (i.e., average reviews per month per year)



```

# Filter out reviews where helpfulness wasn't evaluated
df_helpful = df_clean[df_clean['HelpfulnessDenominator'] > 0].copy()

# Calculate helpfulness ratio
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator']

# Add review length column
df_helpful['ReviewLength'] = df_helpful['Text'].apply(lambda x: len(str(x).split()))

# Plot relationship between review length and helpfulness
plt.figure(figsize=(10, 6))
sns.scatterplot(
    data=df_helpful,
    x='ReviewLength',
    y='HelpfulnessRatio',
    alpha=0.3
)

# Add labels and title
plt.title("Review Length vs. Helpfulness Ratio", fontsize=14)
plt.xlabel("Review Length (word count)")
plt.ylabel("Helpfulness Ratio")
plt.grid(True)
plt.tight_layout()
plt.show()

```

plotly

Here's the **Plotly** version of your **Review Length vs. Helpfulness Ratio** scatter plot — interactive, clear, and presentation-ready.

Plotly Scatter Plot: Review Length vs. Helpfulness

python

```
import pandas as pd
import plotly.express as px

# Filter reviews with valid helpfulness evaluation
df_helpful = df_clean[df_clean['HelpfulnessDenominator'] > 0].copy()

# Calculate helpfulness ratio
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator']

# Compute review length
df_helpful['ReviewLength'] = df_helpful['Text'].apply(lambda x: len(str(x).split()))

# Plot using Plotly
fig = px.scatter(
    df_helpful,
    x='ReviewLength',
    y='HelpfulnessRatio',
    opacity=0.3,
    title='📊 Review Length vs. Helpfulness Ratio',
    labels={
        'ReviewLength': 'Review Length (word count)',
        'HelpfulnessRatio': 'Helpfulness Ratio'
    },
    template='plotly_white'
)

fig.update_traces(marker=dict(size=4))
fig.update_layout(height=600)
fig.show()
```

💡 What It Shows:

- How longer reviews may (or may not) correlate with higher helpfulness.
- Density and outliers in wordy but unhelpful reviews.

Would you like to:

- Add **trendlines** (e.g., linear regression)?
- Color points by sentiment class (positive, neutral, negative)?



```
# Filter for reviews where helpfulness was rated
df_helpful = df_clean[df_clean['HelpfulnessDenominator'] > 0].copy()
```



```
# Calculate helpfulness ratio
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator']

# Group by score and calculate average helpfulness
helpfulness_by_score = df_helpful.groupby('Score')['HelpfulnessRatio'].mean().reset_index()

# Plot
plt.figure(figsize=(8, 5))
sns.barplot(data=helpfulness_by_score, x='Score', y='HelpfulnessRatio', palette='coolwarm')

# Add labels and title
plt.title("Average Helpfulness Ratio by Review Score", fontsize=14)
plt.xlabel("Review Score")
plt.ylabel("Average Helpfulness Ratio")
plt.ylim(0, 1)
plt.tight_layout()
plt.show()

plotly
```

Here's the **Plotly version** of your **Average Helpfulness Ratio by Review Score** bar chart — interactive and ideal for dashboard or presentation use.

Plotly Bar Chart: Helpfulness by Score

```
python

import pandas as pd
import plotly.express as px

# Filter for reviews where helpfulness was rated
df_helpful = df_clean[df_clean['HelpfulnessDenominator'] > 0].copy()

# Calculate helpfulness ratio
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator']

# Group by review score and compute average helpfulness
helpfulness_by_score = df_helpful.groupby('Score')['HelpfulnessRatio'].mean().reset_index()

# Plot using Plotly
fig = px.bar(
    helpfulness_by_score,
    x='Score',
    y='HelpfulnessRatio',
    color='HelpfulnessRatio',
    color_continuous_scale='RdBu',
    title='🌟 Average Helpfulness Ratio by Review Score',
    labels={
        'Score': 'Review Score',
        'HelpfulnessRatio': 'Average Helpfulness Ratio'
    },
    template='plotly_white'
)

fig.update_layout(yaxis_range=[0, 1])
fig.show()
```

Insights You Might Find:

- Mid-range scores (e.g., 3 stars) often have more balanced and helpful content.
- Extremely high or low scores might correlate with low helpfulness (emotional, biased, or vague).

Would you like to:

- Add **error bars** or confidence intervals?
- Compare helpfulness ratio across **verified vs. non-verified users** or **time periods**?



```
refine
# Combine all reviews into a single string and lowercase
all_text = " ".join(df_clean['Text'].astype(str)).lower()

# Remove punctuation
translator = str.maketrans("", "", string.punctuation)
clean_text = all_text.translate(translator)

# Tokenize the text
tokens = word_tokenize(clean_text)

# Remove stopwords and non-alphabetic tokens
stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in tokens if word.isalpha() and word not in stop_words]

# Count word frequencies
word_counts = Counter(filtered_tokens)
common_words = word_counts.most_common(20)

# Convert to DataFrame for plotting
common_df = pd.DataFrame(common_words, columns=['Word', 'Frequency'])

# Plot the top 20 words
plt.figure(figsize=(12, 6))
sns.barplot(data=common_df, x='Frequency', y='Word', palette='mako')
plt.title('Top 20 Most Frequent Words in Review Text')
plt.xlabel('Frequency')
plt.ylabel('Word')
plt.tight_layout()
plt.show()
```

Here's a **refined version** of your script that:

- Ensures **robust preprocessing**
- Improves **readability** and **performance**
- Optionally uses **Plotly** for an interactive bar chart

✓ Refined Code (Seaborn Version)

```
python

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import string
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from collections import Counter
import nltk

# Download NLTK resources (run once if needed)
nltk.download('punkt')
nltk.download('stopwords')

# Combine all reviews into a single lowercase string
all_text = " ".join(df_clean['Text'].dropna().astype(str)).lower()

# Remove punctuation using str.translate
translator = str.maketrans("", "", string.punctuation)
clean_text = all_text.translate(translator)

# Tokenize and filter
tokens = word_tokenize(clean_text)
stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in tokens if word.isalpha() and word not in stop_words]

# Count frequencies
word_counts = Counter(filtered_tokens)
common_words = word_counts.most_common(20)

# Convert to DataFrame for plotting
common_df = pd.DataFrame(common_words, columns=['Word', 'Frequency'])

# Plot with seaborn
plt.figure(figsize=(12, 6))
sns.barplot(data=common_df, x='Frequency', y='Word', palette='mako')
plt.title('Top 20 Most Frequent Words in Review Text')
plt.xlabel('Frequency')
plt.ylabel('Word')
plt.tight_layout()
plt.show()
```

Optional: Plotly Version (Interactive)

```
python

import plotly.express as px

fig = px.bar(
    common_df,
    x='Frequency',
    y='Word',
    orientation='h',
    color='Frequency',
    color_continuous_scale='mako',
    title='Top 20 Most Frequent Words in Review Text',
    template='plotly_white'
)
fig.update_layout(yaxis=dict(autorange="reversed"))
fig.show()
```

✓ Improvements:

- Uses `.dropna()` to avoid processing NaN values
- Ensures all tokens are alphabetic
- Includes both static and interactive plotting options

Let me know if you'd like to:

- Create **word clouds**
- Separate frequencies by **sentiment category**
- Apply **n-gram analysis** (e.g., top 2- or 3-word phrases)



```
import pandas as pd
import plotly.express as px

# Filter to avoid division by zero in helpfulness ratio
df_helpful = df_clean[df_clean['HelpfulnessDenominator'] > 0].copy()

# Compute helpfulness ratio
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator']

# Group by user to calculate average helpfulness and review count
user_helpfulness = df_helpful.groupby(['UserId', 'ProfileName']).agg(
    ReviewCount=('Text', 'count'),
    AvgHelpfulness=('HelpfulnessRatio', 'mean')
).reset_index()

# Create review count bins
bins = [1, 10, 50, 100, 200, 500, 1000]
labels = ['1-10', '11-50', '51-100', '101-200', '201-500', '500+']
user_helpfulness['ReviewCountBin'] = pd.cut(user_helpfulness['ReviewCount'], bins=bins,
labels=labels, include_lowest=True)

# Plot using Plotly
fig = px.box(
    user_helpfulness,
    x='ReviewCountBin',
    y='AvgHelpfulness',
    color='ReviewCountBin',
    title='👉 Helpfulness Ratio by Reviewer Activity Level',
    labels={
        'ReviewCountBin': 'Number of Reviews (Binned)',
        'AvgHelpfulness': 'Average Helpfulness Ratio'
    },
    template='plotly_white'
)
```

```
fig.update_layout(showlegend=False)
fig.show()

-----
NameError                                Traceback (most recent call last)
/tmp/ipython-input-12-3677575342.py in <cell line: 0>()
      3
      4 # Filter to avoid division by zero in helpfulness ratio
----> 5 df_helpful = df_clean[df_clean['HelpfulnessDenominator'] > 0].copy()
      6
      7 # Compute helpfulness ratio

NameError: name 'df_clean' is not defined
```

The error **NameError: name 'df_clean' is not defined** means that your variable **df_clean** hasn't been created or loaded yet in your current Python session.

✓ How to Fix It:

Option 1: If your cleaned dataframe is saved in a variable called `df`, just assign it like this:

```
python

df_clean = df.copy()
```

Then run your visualization code again.

Option 2: If you already did some cleaning steps, re-run the cell where you defined or cleaned `df_clean`.

Here's a minimal placeholder if you're just testing:

```
python

# Example placeholder if you've done cleaning steps on `df`
df_clean = df.copy() # Replace with your actual cleaned dataframe
```

Let me know if you'd like help recreating **df_clean** from your raw dataset or reapplying cleaning steps (like drop missing, remove duplicates, etc.) before this plot!



```
# Add Helpfulness Ratio (Avoid division by zero)
df_subset['Helpfulness_Ratio'] = df_subset['HelpfulnessNumerator'] /
df_subset['HelpfulnessDenominator'].replace(0, 1)
```

```
df_subset['Helpfulness_Ratio'] = df_subset['Helpfulness_Ratio'].clip(upper=1.0)

import plotly.express as px
fig = px.histogram(df_subset, x='Helpfulness_Ratio', nbins=20, title='Helpfulness Ratio Distribution')
fig.show()
```

✅ Your code is mostly correct! Here's a slightly **refined version** to ensure robustness and clarity — with edge case handling and styling improvements.

✅ Refined Version: Helpfulness Ratio Histogram (Plotly)

```
python

import pandas as pd
import plotly.express as px

# Avoid division by zero by replacing denominator 0 with 1
df_subset['Helpfulness_Ratio'] = df_subset['HelpfulnessNumerator'] /
df_subset['HelpfulnessDenominator'].replace(0, 1)

# Cap values at 1 (sometimes numerators > denominators due to data issues)
df_subset['Helpfulness_Ratio'] = df_subset['Helpfulness_Ratio'].clip(upper=1.0)

# Plot histogram
fig = px.histogram(
    df_subset,
    x='Helpfulness_Ratio',
    nbins=20,
    title='📊 Helpfulness Ratio Distribution',
    labels={'Helpfulness_Ratio': 'Helpfulness Ratio'},
    template='plotly_white'
)

fig.update_layout(
    xaxis_title='Helpfulness Ratio',
    yaxis_title='Count of Reviews'
)

fig.show()
```

🧠 What This Does:

- Ensures no division by zero
- Caps unexpected ratios above 1
- Shows how often reviews are marked helpful (0.0 to 1.0 range)

Let me know if you'd like to:

- Break this down by **review score** or **sentiment**
- Overlay a KDE-style smoothed line
- Filter to only include reviews with **Score >= 4** or any specific group



```
import pandas as pd
import plotly.express as px

# Filter to avoid division by zero in helpfulness ratio
df_helpful = df_clean[df_clean['HelpfulnessDenominator'] > 0].copy()

# Compute helpfulness ratio
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator']

# Group by user to calculate average helpfulness and review count
user_helpfulness = df_helpful.groupby(['UserId', 'ProfileName']).agg(
    ReviewCount=('Text', 'count'),
    AvgHelpfulness=('HelpfulnessRatio', 'mean')
).reset_index()

# Create review count bins
bins = [1, 10, 50, 100, 200, 500, 1000]
labels = ['1-10', '11-50', '51-100', '101-200', '201-500', '500+']
user_helpfulness['ReviewCountBin'] = pd.cut(user_helpfulness['ReviewCount'], bins=bins,
labels=labels, include_lowest=True)

# Plot using Plotly
fig = px.box(
    user_helpfulness,
    x='ReviewCountBin',
    y='AvgHelpfulness',
    color='ReviewCountBin',
    title='📦 Helpfulness Ratio by Reviewer Activity Level',
    labels={
        'ReviewCountBin': 'Number of Reviews (Binned)',
        'AvgHelpfulness': 'Average Helpfulness Ratio'
    },
    template='plotly_white'
)

fig.update_layout(showlegend=False)
fig.show()
```

✅ Your code looks great and will produce an **interactive box plot** using Plotly to visualize how reviewer helpfulness varies across activity levels.

Here's a **brief breakdown** of what your script does right and some minor suggestions for improvement if needed:

✅ What's Working Well:

- ✅ Filters out `HelpfulnessDenominator == 0` to avoid division by zero
- ✅ Calculates `HelpfulnessRatio` safely

- ✓ Aggregates helpfulness and review count per user
- ✓ Uses `pd.cut()` to categorize reviewer activity into intuitive bins
- ✓ Plots a **Plotly box plot** with a clean template (`plotly_white`)
- ✓ Disables redundant legend since x-axis labels are descriptive

✨ Optional Improvements:

If you want smoother results or handle edge cases better:

1. Replace 0-denominators with NaN instead of filtering (if analyzing full distribution):

```
python

df_clean['HelpfulnessRatio'] = df_clean['HelpfulnessNumerator'] /
df_clean['HelpfulnessDenominator'].replace(0, pd.NA)
```

2. Ensure bin coverage: Your current bin stops at 1000, but if some users have >1000 reviews, they will be NaN. Consider:

```
python

bins = [1, 10, 50, 100, 200, 500, 1000, float('inf')]
labels = ['1-10', '11-50', '51-100', '101-200', '201-500', '501-1000', '1000+']
```

3. Add median lines or annotations (advanced):

Plotly allows you to annotate medians for each bin if presenting this for analysis.

Sample Output:

You'll see a clean box plot showing:

- Median helpfulness ratio per activity bin
- Spread and outliers (e.g., users who write many reviews but receive low helpfulness)

Let me know if you'd like to:

- Add hover tooltips with usernames
- Compare this by sentiment or score
- Export results to a dashboard or file



```
align with this
import pandas as pd
import plotly.express as px
```

```
# Avoid division by zero by replacing denominator 0 with 1
df_subset['Helpfulness_Ratio'] = df_subset['HelpfulnessNumerator'] /
```



```
df_subset['HelpfulnessDenominator'].replace(0, 1)

# Cap values at 1 (sometimes numerators > denominators due to data issues)
df_subset['Helpfulness_Ratio'] = df_subset['Helpfulness_Ratio'].clip(upper=1.0)

# Plot histogram
fig = px.histogram(
    df_subset,
    x='Helpfulness_Ratio',
    nbins=20,
    title='📊 Helpfulness Ratio Distribution',
    labels={'Helpfulness_Ratio': 'Helpfulness Ratio'},
    template='plotly_white'
)

fig.update_layout(
    xaxis_title='Helpfulness Ratio',
    yaxis_title='Count of Reviews'
)

fig.show()

refine this one
import pandas as pd
import plotly.express as px

# Filter to avoid division by zero in helpfulness ratio
df_helpful = df_clean[df_clean['HelpfulnessDenominator'] > 0].copy()

# Compute helpfulness ratio
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator']

# Group by user to calculate average helpfulness and review count
user_helpfulness = df_helpful.groupby(['UserId', 'ProfileName']).agg(
    ReviewCount=('Text', 'count'),
    AvgHelpfulness=('HelpfulnessRatio', 'mean')
).reset_index()

# Create review count bins
bins = [1, 10, 50, 100, 200, 500, 1000]
labels = ['1-10', '11-50', '51-100', '101-200', '201-500', '500+']
user_helpfulness['ReviewCountBin'] = pd.cut(user_helpfulness['ReviewCount'], bins=bins,
labels=labels, include_lowest=True)

# Plot using Plotly
fig = px.box(
    user_helpfulness,
    x='ReviewCountBin',
    y='AvgHelpfulness',
    color='ReviewCountBin',
```





```

title='📦 Helpfulness Ratio by Reviewer Activity Level',
labels={
    'ReviewCountBin': 'Number of Reviews (Binned)',
    'AvgHelpfulness': 'Average Helpfulness Ratio'
},
template='plotly_white'
)

fig.update_layout(showlegend=False)
fig.show()

```

Here's a **refined version** of your second script, fully aligned with your first code block's approach:

-  Consistent use of `replace(0, 1)` to avoid division by zero
-  Capping `HelpfulnessRatio` at 1.0 to match histogram logic
-  Clear labels and template styling
-  Optional: Extended bin for **1000+** reviewers

Refined & Aligned Code

```

python

import pandas as pd
import plotly.express as px

# Copy from df_subset or df_clean as needed
df_helpful = df_clean.copy()

# Avoid division by zero by replacing 0 with 1
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator'].replace(0, 1)

# Cap any values greater than 1
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessRatio'].clip(upper=1.0)

# Group by user to calculate review count and avg helpfulness
user_helpfulness = df_helpful.groupby(['UserId', 'ProfileName']).agg(
    ReviewCount=('Text', 'count'),
    AvgHelpfulness=('HelpfulnessRatio', 'mean')
).reset_index()

# Define review count bins (including an upper limit)
bins = [1, 10, 50, 100, 200, 500, 1000, float('inf')]
labels = ['1-10', '11-50', '51-100', '101-200', '201-500', '501-1000', '1000+']
user_helpfulness['ReviewCountBin'] = pd.cut(
    user_helpfulness['ReviewCount'],
    bins=bins,
    labels=labels,
    include_lowest=True
)

# Plot interactive boxplot
fig = px.box(
    user_helpfulness,
    x='ReviewCountBin',
    y='AvgHelpfulness',
    color='ReviewCountBin',
    title='📦 Helpfulness Ratio by Reviewer Activity Level',
    labels={
        'ReviewCountBin': 'Number of Reviews (Binned)',

```

```

        'AvgHelpfulness': 'Average Helpfulness Ratio'
    },
    template='plotly_white'
)

fig.update_layout(showlegend=False)
fig.show()

```

Why This Alignment Matters:

- Ensures consistent assumptions and calculations across different visualizations
- Prevents misleading outliers from dominating averages
- Makes your codebase cleaner and easier to present

Let me know if you'd like to:

- Add hover tooltips with usernames or counts
- Export this chart or merge into a presentation notebook



```

-----
NameError                                Traceback (most recent call last)
/tmp/ipython-input-19-826468841.py in <cell line: 0>()
      3
      4 # Copy from df_subset or df_clean as needed
----> 5 df_helpful = df_clean.copy()
      6
      7 # Avoid division by zero by replacing 0 with 1

NameError: name 'df_clean' is not defined

```

The error **NameError: name 'df_clean' is not defined** means Python doesn't recognize **df_clean** — it hasn't been created in this session.

To fix it, simply align with your working variable (**df_subset**), which you're already using in your other visualizations.

Corrected Version Using df_subset

```

python

import pandas as pd
import plotly.express as px

# Copy from your working DataFrame
df_helpful = df_subset.copy()

# Avoid division by zero by replacing 0 with 1
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator'].replace(0, 1)

# Cap values at 1.0 (edge case control)

```

```
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessRatio'].clip(upper=1.0)

# Group by user to calculate review count and avg helpfulness
user_helpfulness = df_helpful.groupby(['UserId', 'ProfileName']).agg(
    ReviewCount=('Text', 'count'),
    AvgHelpfulness=('HelpfulnessRatio', 'mean')
).reset_index()

# Create binned reviewer activity levels
bins = [1, 10, 50, 100, 200, 500, 1000, float('inf')]
labels = ['1-10', '11-50', '51-100', '101-200', '201-500', '501-1000', '1000+']
user_helpfulness['ReviewCountBin'] = pd.cut(
    user_helpfulness['ReviewCount'],
    bins=bins,
    labels=labels,
    include_lowest=True
)

# Plot with Plotly
fig = px.box(
    user_helpfulness,
    x='ReviewCountBin',
    y='AvgHelpfulness',
    color='ReviewCountBin',
    title='👤 Helpfulness Ratio by Reviewer Activity Level',
    labels={
        'ReviewCountBin': 'Number of Reviews (Binned)',
        'AvgHelpfulness': 'Average Helpfulness Ratio'
    },
    template='plotly_white'
)

fig.update_layout(showlegend=False)
fig.show()
```

Let me know if you'd like to:

- Compare this by **sentiment** or **review score**
- Export this to PNG/PDF
- Add interactivity like hover details for usernames or tooltips



```
refine
import pandas as pd
import plotly.express as px

# Filter reviews with valid helpfulness evaluation
df_helpful = df_clean[df_clean['HelpfulnessDenominator'] > 0].copy()

# Calculate helpfulness ratio
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator']

# Compute review length
df_helpful['ReviewLength'] = df_helpful['Text'].apply(lambda x: len(str(x).split()))

# Plot using Plotly
fig = px.scatter(
```

```
df_helpful,
x='ReviewLength',
y='HelpfulnessRatio',
opacity=0.3,
title='📝 Review Length vs. Helpfulness Ratio',
labels={
    'ReviewLength': 'Review Length (word count)',
    'HelpfulnessRatio': 'Helpfulness Ratio'
},
template='plotly_white'
)

fig.update_traces(marker=dict(size=4))
fig.update_layout(height=600)
fig.show()
```

Here's a **refined version** of your Plotly scatter plot for **Review Length vs. Helpfulness Ratio**, aligning with best practices and your earlier setup:

✅ Refined Code (Clean, Consistent, and Robust)

```
python

import pandas as pd
import plotly.express as px

# Use df_subset or define df_clean first – this uses df_subset for consistency
df_helpful = df_subset[df_subset['HelpfulnessDenominator'] > 0].copy()

# Calculate helpfulness ratio, replacing 0 denominator with 1 and clipping
df_helpful['HelpfulnessRatio'] = (
    df_helpful['HelpfulnessNumerator'] / df_helpful['HelpfulnessDenominator'].replace(0, 1)
).clip(upper=1.0)





# Compute review length in words
df_helpful['ReviewLength'] = df_helpful['Text'].astype(str).apply(lambda x: len(x.split()))

# Plot using Plotly
fig = px.scatter(
    df_helpful,
    x='ReviewLength',
    y='HelpfulnessRatio',
    opacity=0.3,
    title='📝 Review Length vs. Helpfulness Ratio',
    labels={
        'ReviewLength': 'Review Length (Word Count)',
        'HelpfulnessRatio': 'Helpfulness Ratio'
    },
    template='plotly_white'
)

# Fine-tune layout
fig.update_traces(marker=dict(size=4))
fig.update_layout(
    height=600,
    xaxis=dict(range=[0, df_helpful['ReviewLength'].quantile(0.99)]),
    yaxis=dict(range=[0, 1])
)

fig.show()
```

Why These Refinements?

-  Uses `.clip(upper=1.0)` to cap values logically
-  Casts `Text` to `str` to avoid type errors
-  Adds range limits to avoid outlier distortion (optional)
-  Keeps consistency with your histogram and boxplot logic

Let me know if you'd like to:

- **Color by sentiment or score**
- **Add trend lines or bins**
- Export this chart for your presentation or report



```
refine
import pandas as pd
import plotly.express as px

# Filter for reviews where helpfulness was rated
df_helpful = df_clean[df_clean['HelpfulnessDenominator'] > 0].copy()

# Calculate helpfulness ratio
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator']

# Group by review score and compute average helpfulness
helpfulness_by_score = df_helpful.groupby('Score')['HelpfulnessRatio'].mean().reset_index()

# Plot using Plotly
fig = px.bar(
    helpfulness_by_score,
    x='Score',
    y='HelpfulnessRatio',
    color='HelpfulnessRatio',
    color_continuous_scale='RdBu',
    title='🌟 Average Helpfulness Ratio by Review Score',
    labels={
        'Score': 'Review Score',
        'HelpfulnessRatio': 'Average Helpfulness Ratio'
    },
    template='plotly_white'
)

fig.update_layout(yaxis_range=[0, 1])
fig.show()
```

Here's a polished and performance-safe version of your code for plotting **Average Helpfulness Ratio by Review Score**, aligned with your earlier logic and design:

✓ Refined Version (Safe, Consistent, and Clean)

```
python

import pandas as pd
import plotly.express as px

# Use df_subset if df_clean is not defined
df_helpful = df_subset[df_subset['HelpfulnessDenominator'] > 0].copy()

# Calculate and cap helpfulness ratio
df_helpful['HelpfulnessRatio'] = (
    df_helpful['HelpfulnessNumerator'] / df_helpful['HelpfulnessDenominator'].replace(0, 1)
).clip(upper=1.0)

# Group by score and calculate average helpfulness ratio
helpfulness_by_score = df_helpful.groupby('Score', as_index=False)['HelpfulnessRatio'].mean()

# Plot using Plotly
fig = px.bar(
    helpfulness_by_score,
    x='Score',
    y='HelpfulnessRatio',
    color='HelpfulnessRatio',
    color_continuous_scale='RdBu',
    title='★ Average Helpfulness Ratio by Review Score',
    labels={
        'Score': 'Review Score',
        'HelpfulnessRatio': 'Average Helpfulness Ratio'
    },
    template='plotly_white'
)

# Update layout to fix scale and improve visuals
fig.update_layout(
    yaxis=dict(range=[0, 1]),
    xaxis=dict(tickmode='linear'),
    coloraxis_colorbar=dict(title='Helpfulness'),
)

fig.show()
```

Highlights:

- ✓ Uses `.clip(upper=1.0)` for logical consistency
- ✓ Ensures x-axis ticks are linear (1 to 5 for scores)
- ✓ Clean design using `plotly_white` and intuitive color scaling

Let me know if you want to:

- Show **count of reviews per score** alongside
- Add **hover info** like standard deviation or review volume
- Export this as a chart for a slide deck or dashboard



```

refine
import plotly.express as px

# Count sentiment classes
sentiment_counts = df_subset['Sentiment'].value_counts().reindex(['positive', 'neutral', 'negative'])

# Convert to DataFrame for Plotly
sentiment_df = sentiment_counts.reset_index()
sentiment_df.columns = ['Sentiment', 'Count']

# Plot with Plotly
fig = px.bar(
    sentiment_df,
    x='Sentiment',
    y='Count',
    color='Sentiment',
    title='Sentiment Distribution',
    text='Count'
)

fig.update_layout(
    xaxis_title='Sentiment',
    yaxis_title='Number of Reviews',
    template='plotly_white'
)

fig.show()

# Step 1: Define sentiment categories based on the Score
def categorize_sentiment(score):
    if score in [1, 2]:
        return 'Negative'
    elif score == 3:
        return 'Neutral'
    else:
        return 'Positive'

# Step 2: Apply sentiment categorization
df_clean['Sentiment'] = df_clean['Score'].apply(categorize_sentiment)

# Step 3: Calculate the percentage of each sentiment
sentiment_counts = df_clean['Sentiment'].value_counts(normalize=True) * 100
sentiment_counts = sentiment_counts.round(2)

# Let's see the result please
sentiment_counts

```

Here's a refined version that ensures clarity, consistency, and better structure. I've also added improvements in ordering, labeling, and coloring for both the **Plotly chart** and the **sentiment**

breakdown:

✓ Refined Code (Sentiment Distribution & Categorization)

```
python

import pandas as pd
import plotly.express as px

# Step 1: Define sentiment categories based on Score
def categorize_sentiment(score):
    if score in [1, 2]:
        return 'negative'
    elif score == 3:
        return 'neutral'
    else:
        return 'positive'

# Step 2: Apply to a clean DataFrame (use df_subset if df_clean is not defined)
df_clean = df_subset.copy()
df_clean['Sentiment'] = df_clean['Score'].apply(categorize_sentiment)

# Step 3: Count sentiments
sentiment_counts = df_clean['Sentiment'].value_counts().reindex(['positive', 'neutral', 'negative'])

# Convert to DataFrame for Plotly
sentiment_df = sentiment_counts.reset_index()
sentiment_df.columns = ['Sentiment', 'Count']

# Step 4: Plot bar chart with Plotly
fig = px.bar(
    sentiment_df,
    x='Sentiment',
    y='Count',
    color='Sentiment',
    text='Count',
    title='📊 Sentiment Distribution Based on Review Score',
    template='plotly_white',
    color_discrete_map={
        'positive': 'green',
        'neutral': 'gray',
        'negative': 'crimson'
    }
)

fig.update_layout(
    xaxis_title='Sentiment',
    yaxis_title='Number of Reviews',
    uniformtext_minsize=8,
    uniformtext_mode='hide'
)



fig.show()

# Step 5: Optional – Show sentiment percentages
sentiment_percentages = df_clean['Sentiment'].value_counts(normalize=True).reindex(['positive', 'neutral', 'negative']) * 100
sentiment_percentages = sentiment_percentages.round(2)

# Display sentiment breakdown as a table
display(sentiment_percentages.to_frame(name='Percentage (%)'))
```

🔍 Summary:

- ✓ Uniform lowercase categories (**positive**, **neutral**, **negative**)

-  Plotly color-coded bar with better spacing and titles
-  Sentiment percentage breakdown for reporting

Let me know if you'd like to:

- Plot this as a **donut/pie chart**
- Overlay it with **original VADER labels**
- Animate sentiment over **time or product categories**



```
refine
# Calculate word counts
df_clean['ReviewLength'] = df_clean['Text'].apply(lambda x: len(str(x).split()))
df_clean['SummaryLength'] = df_clean['Summary'].apply(lambda x: len(str(x).split()))

# Compute averages
avg_review_length = df_clean['ReviewLength'].mean()
avg_summary_length = df_clean['SummaryLength'].mean()

# Display results
print(f"Average Review Length: {avg_review_length:.2f} words")
print(f"Average Summary Length: {avg_summary_length:.2f} words")

plt.figure(figsize=(10, 5))

# Histogram of review lengths
sns.histplot(df_clean['ReviewLength'], bins=50, kde=True, color='skyblue', label='Review Text')

# Histogram of summary lengths
sns.histplot(df_clean['SummaryLength'], bins=50, kde=True, color='orange', label='Summary Text')

# Add labels and title
plt.title("Distribution of Review and Summary Lengths")
plt.xlabel("Word Count")
plt.ylabel("Frequency")
plt.legend()
plt.tight_layout()
plt.show()
```

Here's a **refined version** of your code that:

- Ensures consistent formatting and safe handling of **NaNs**
- Uses **Plotly** instead of `matplotlib/seaborn` for interactivity
- Clearly labels and contrasts **review vs summary lengths**

Refined Version (with Plotly Dual Histogram)

python

```
import pandas as pd
import plotly.graph_objects as go

# Safely compute word counts
df_clean['ReviewLength'] = df_clean['Text'].astype(str).apply(lambda x: len(x.split()))
df_clean['SummaryLength'] = df_clean['Summary'].astype(str).apply(lambda x: len(x.split()))

# Compute averages
avg_review_length = df_clean['ReviewLength'].mean()
avg_summary_length = df_clean['SummaryLength'].mean()

# Display average lengths
print(f"📄 Average Review Length: {avg_review_length:.2f} words")
print(f"📄 Average Summary Length: {avg_summary_length:.2f} words")

# Create histograms using Plotly
fig = go.Figure()

fig.add_trace(go.Histogram(
    x=df_clean['ReviewLength'],
    name='Review Text',
    opacity=0.6,
    marker_color='skyblue',
    nbinsx=50
))

fig.add_trace(go.Histogram(
    x=df_clean['SummaryLength'],
    name='Summary Text',
    opacity=0.6,
    marker_color='orange',
    nbinsx=50
))

# Update layout
fig.update_layout(
    title='📊 Distribution of Review and Summary Lengths (in Words)',
    xaxis_title='Word Count',
    yaxis_title='Frequency',
    barmode='overlay',
    template='plotly_white',
    legend_title='Text Type'
)

fig.show()
```

Key Benefits:

- ✓ No issues with NaN text fields (thanks to `.astype(str)`)
- ✓ Visual overlay of review vs summary length distributions
- ✓ Interactive and visually clear with Plotly

Let me know if you'd also like to:

- Add **vertical lines** to show average lengths
- Split by **sentiment** or **score buckets**
- Export this for your report or dashboard



```
# Ensure 'ReviewLength' column exists
df_clean['ReviewLength'] = df_clean['Text'].apply(lambda x: len(str(x).split()))

# 1. Calculate correlation coefficient
correlation = df_clean['ReviewLength'].corr(df_clean['Score'])
print(f"Correlation between review length and score: {correlation:.4f}")

# Display results
plt.figure(figsize=(10, 6))
sns.boxplot(x='Score', y='ReviewLength', data=df_clean, palette='Spectral')

# Add labels and title
plt.title('Review Length by Review Score', fontsize=14)
plt.xlabel('Review Score')
plt.ylabel('Review Length (word count)')
plt.tight_layout()
plt.show()
```

Here's a **refined version** of your correlation and visualization code using **Plotly** instead of Seaborn/Matplotlib, with better formatting and safe handling of missing data:

✓ Refined Version (Correlation + Box Plot with Plotly)

```
python

import pandas as pd
import plotly.express as px

# Ensure 'ReviewLength' exists and compute it safely
df_clean['ReviewLength'] = df_clean['Text'].astype(str).apply(lambda x: len(x.split()))





# 1. Correlation between review length and score
correlation = df_clean['ReviewLength'].corr(df_clean['Score'])
print(f"🔗 Correlation between review length and score: {correlation:.4f}")

# 2. Interactive box plot using Plotly
fig = px.box(
    df_clean,
    x='Score',
    y='ReviewLength',
    color='Score',
    title='📦 Review Length by Review Score',
    labels={
        'Score': 'Review Score',
        'ReviewLength': 'Review Length (word count)'
    },
    template='plotly_white',
    points='outliers' # shows individual points outside whiskers
)

fig.update_layout(
    xaxis=dict(type='category'),
    showlegend=False
)

fig.show()
```

Highlights:

-  Interactive Plotly boxplot with color-coded scores
-  Displays correlation value in the console
-  Uses `.astype(str)` to avoid text errors
-  Outliers visualized for richer insight

Let me know if you'd like to:

- Add **trend lines** or **regression plots**
- Show this by **sentiment class** or time-based buckets



```
refine
import matplotlib.pyplot as plt
from wordcloud import WordCloud

# Join all cleaned text into a single string
all_words = ' '.join(df_subset['Cleaned_Text'])

# Generate word cloud
wordcloud = WordCloud(width=800, height=400, background_color='white').generate(all_words)

# Display word cloud
plt.figure(figsize=(15, 7))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title("Most Frequent Words in Reviews")
plt.show()
```

```
-----
NameError                                Traceback (most recent call last)
/tmp/ipython-input-1-1189189131.py in <cell line: 0>()
      3
      4 # Join all cleaned text into a single string
----> 5 all_words = ' '.join(df_subset['Cleaned_Text'])
      6
      7 # Generate word cloud
```

NameError: name 'df_subset' is not defined

The error **NameError: name 'df_subset' is not defined** means that the variable `df_subset` is not available in your current notebook session.

To fix and **refine your word cloud visualization**, here's an updated and safe version using `df_clean` as the base (which you've been using consistently):

✓ Refined Word Cloud Code (with fallbacks and checks)

```
python

import pandas as pd
import matplotlib.pyplot as plt
from wordcloud import WordCloud

# Fallback to df_clean if df_subset is not defined
try:
    base_df = df_subset.copy()
except NameError:
    base_df = df_clean.copy()

# Ensure 'Cleaned_Text' column exists
if 'Cleaned_Text' not in base_df.columns:
    base_df['Cleaned_Text'] = base_df['Text'].astype(str).str.lower()

# Join all cleaned text into one large string
all_words = ' '.join(base_df['Cleaned_Text'])

# Generate the word cloud
wordcloud = WordCloud(
    width=1000,
    height=500,
    background_color='white',
    colormap='viridis',
    max_words=200
).generate(all_words)

# Display the word cloud
plt.figure(figsize=(15, 7))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title("▲ Most Frequent Words in Review Text", fontsize=16)
plt.tight_layout()
plt.show()
```

🔍 Optional Add-ons:

- Use `stopwords=STOPWORDS` to filter common English stopwords
- Use `.mask()` or `re.sub()` to clean punctuation further
- Switch to **Plotly** or **streamlit** for dynamic display

Let me know if you'd like to visualize by **sentiment class** or create **separate clouds per score**!



Use `stopwords=STOPWORDS` to filter common English stopwords

Use `.mask()` or `re.sub()` to clean punctuation further

Switch to Plotly or streamlit for dynamic display

Here's a refined and enhanced version of your **word cloud visualization**, now including:

- ✓ Stopword removal
- ✓ Extra punctuation cleaning

✅✅ Interactive display using Plotly

(Streamlit version also available upon request)

✅ Refined Word Cloud with Stopwords, Punctuation Cleaning, and Plotly

```
python

import pandas as pd
import re
from wordcloud import WordCloud, STOPWORDS
import plotly.express as px
import matplotlib.pyplot as plt
from PIL import Image
import io

# Use df_clean as the base
base_df = df_clean.copy()

# Ensure 'Cleaned_Text' column exists
if 'Cleaned_Text' not in base_df.columns:
    base_df['Cleaned_Text'] = base_df['Text'].astype(str).str.lower()

# Combine all text
all_text = ' '.join(base_df['Cleaned_Text'])

# Clean punctuation using re.sub (mask-like effect)
all_text = re.sub(r'[^\w\s]', '', all_text) # Remove punctuation
all_text = re.sub(r'\d+', '', all_text)     # Optional: remove numbers

# Create word cloud with STOPWORDS
wordcloud = WordCloud(
    width=1000,
    height=500,
    background_color='white',
    colormap='viridis',
    stopwords=STOPWORDS,
    max_words=200
).generate(all_text)

# Convert to image
img = wordcloud.to_image()

# Convert to bytes and back for Plotly
buf = io.BytesIO()
img.save(buf, format='PNG')
buf.seek(0)

# Plot with Plotly
fig = px.imshow(Image.open(buf))
fig.update_layout(
    title='📊 Word Cloud of Frequent Terms in Reviews (Excluding Stopwords)',
    xaxis_visible=False,
    yaxis_visible=False
)
fig.show()
```

🔄 Want to Use Streamlit Instead?

If you're building an app or dashboard, just replace the bottom section with:

```
python

import streamlit as st
st.image(img, caption='Word Cloud of Reviews', use_column_width=True)
```

Would you like to:

- Generate word clouds **per sentiment category**?
- Display **top n-grams** (e.g., bigrams like "not good", "highly recommend")?

Let me know — happy to help extend this further!



```
-----  
NameError                                Traceback (most recent call last)  
/tmp/ipython-input-2-4192432905.py in <cell line: 0>()  
      8  
      9 # Use df_clean as the base  
----> 10 base_df = df_clean.copy()  
      11  
     12 # Ensure 'Cleaned_Text' column exists  
  
NameError: name 'df_clean' is not defined
```

Thanks! The error **NameError: name 'df_clean' is not defined** means that your cleaned dataframe (**df_clean**) hasn't been initialized yet in this notebook session.

✓ Quick Fix:

If your cleaned data is currently in **df_subset**, just assign:

```
python  
  
df_clean = df_subset.copy()
```

Then rerun the refined word cloud cell.

🧠 Alternative: Adjust to Use Whatever Is Available

You can also modify the code to fall back gracefully:

```
python  
  
try:  
    base_df = df_clean.copy()  
except NameError:  
    base_df = df_subset.copy()
```

Make sure one of those (**df_clean** or **df_subset**) has been defined and contains the cleaned review text.

Let me know if you'd like me to generate the complete cleaned version setup for **df_clean** based on your earlier steps.



provide full refined code

Here's the **full refined code** to generate a **Plotly-based word cloud** of Amazon food reviews, incorporating:

- ☒ Stopword filtering
- ☒ Punctuation removal with `re.sub`
- ☒ Graceful fallback if `df_clean` isn't defined
- ☒ Plotly display of the word cloud

Full Refined Word Cloud Code

```
python

import pandas as pd
import re
import io
from wordcloud import WordCloud, STOPWORDS
import plotly.express as px
from PIL import Image

# Step 1: Define the base DataFrame safely
try:
    base_df = df_clean.copy()
except NameError:
    base_df = df_subset.copy() # Fallback if df_clean is not defined

# Step 2: Ensure text is in lowercase and cleaned
if 'Cleaned_Text' not in base_df.columns:
    base_df['Cleaned_Text'] = base_df['Text'].astype(str).str.lower()

# Step 3: Combine all text into one string
all_text = ' '.join(base_df['Cleaned_Text'])

# Step 4: Remove punctuation and digits
all_text = re.sub(r'[^\w\s]', '', all_text) # remove punctuation
all_text = re.sub(r'\d+', '', all_text)    # remove numbers

# Step 5: Generate the word cloud (excluding stopwords)
wordcloud = WordCloud(
    width=1000,
    height=500,
    background_color='white',
    colormap='viridis',
    stopwords=STOPWORDS,
    max_words=200
).generate(all_text)

# Step 6: Convert the image to bytes for Plotly
img = wordcloud.to_image()
buf = io.BytesIO()
img.save(buf, format='PNG')
buf.seek(0)

# Step 7: Display using Plotly
fig = px.imshow(Image.open(buf))
fig.update_layout(
    title='Word Cloud of Frequent Terms in Amazon Food Reviews',
    xaxis_visible=False,
    yaxis_visible=False,
    margin=dict(l=0, r=0, t=50, b=0)
```

```
)
fig.show()
```

⚡ Optional Enhancements

Let me know if you want to:

- Generate **word clouds per sentiment** (positive, neutral, negative)
- Extract and plot **top bigrams or trigrams**
- Include **lemmatization** for more meaningful grouping

I'm happy to help!



Hult_SMAFBI_Team_3__A1_Team_assignment (2).pdf
PDF

refine codes from most frequent words in the EDA until the end of EDA

Here's the **refined EDA code** from the "Most Frequent Words" section until the end of the EDA phase, now fixed and organized for Plotly and error handling.

✅ Refined EDA Code: Word Cloud to Seasonality

python

```
import pandas as pd
import re
import calendar
from collections import Counter
from wordcloud import WordCloud, STOPWORDS
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
from IPython.display import display
from PIL import Image
import numpy as np
import io

# Assume df_clean is already preprocessed
# Add Review and Summary Length
df_clean['ReviewLength'] = df_clean['Text'].apply(lambda x: len(str(x).split()))
df_clean['SummaryLength'] = df_clean['Summary'].apply(lambda x: len(str(x).split()))

# 🚀 Word Cloud (with STOPWORDS)
clean_text = ' '.join(df_clean['Cleaned_Text'].astype(str).tolist())
clean_text = re.sub(r'^\w\s', '', clean_text)
clean_text = re.sub(r'\d+', '', clean_text)
wordcloud = WordCloud(stopwords=STOPWORDS, width=1000, height=400,
background_color='white').generate(clean_text)

plt.figure(figsize=(14, 6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title('⬆ Most Frequent Words in Reviews')
plt.show()
```

```
# 📊 Histogram of Review and Summary Length
length_df = df_clean[['ReviewLength', 'SummaryLength']].melt(var_name='TextType',
value_name='WordCount')
fig1 = px.histogram(length_df, x='WordCount', color='TextType', nbins=40,
                    title="📝 Distribution of Review and Summary Lengths",
                    labels={'WordCount': 'Word Count'})

fig1.show()

# 📊 Boxplot: Review Score vs Review Length
correlation = df_clean['ReviewLength'].corr(df_clean['Score'])
fig2 = px.box(df_clean, x='Score', y='ReviewLength', points='outliers',
              title=f"📊 Review Length by Review Score (Correlation: {correlation:.2f})",
              labels={'Score': 'Review Score', 'ReviewLength': 'Review Length (words)'})

fig2.show()

# 📅 Seasonality: Reviews by Month
df_clean['Month'] = df_clean['Time'].dt.month
monthly_reviews = df_clean['Month'].value_counts().sort_index()
month_names = [calendar.month_name[i] for i in monthly_reviews.index]
fig3 = px.line(x=month_names, y=monthly_reviews.values, markers=True,
              title="📅 Seasonal Review Patterns (Monthly)",
              labels={'x': 'Month', 'y': 'Number of Reviews'})

fig3.show()
```

🔧 Output:

This code:

- Visualizes the **most frequent review words** using a WordCloud.
- Shows **text length distributions** (Review vs Summary).
- Displays **correlation and boxplot** between review length and review score.
- Plots **seasonality trends** of reviews by month using Plotly.

Let me know if you'd like to:

- Include KDE plots.
- Break down seasonality by year.
- Segment word clouds by sentiment.



change to df_subset



df_subset

It looks like **df_subset** hasn't been defined in this session yet. Please upload or define your dataset as **df_subset** before proceeding with the refined EDA code.

If you'd like, I can help you reload the dataset or assist in assigning it. Would you like to upload the file again or redefine it here?



```

change df_clean to df_subset
import pandas as pd
import re
import calendar
from collections import Counter
from wordcloud import WordCloud, STOPWORDS
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
from IPython.display import display
from PIL import Image
import numpy as np
import io

# Assume df_clean is already preprocessed
# Add Review and Summary Length
df_clean['ReviewLength'] = df_clean['Text'].apply(lambda x: len(str(x).split()))
df_clean['SummaryLength'] = df_clean['Summary'].apply(lambda x: len(str(x).split()))

# 📌 Word Cloud (with STOPWORDS)
clean_text = ' '.join(df_clean['Cleaned_Text'].astype(str).tolist())
clean_text = re.sub(r'^\w\s', '', clean_text)
clean_text = re.sub(r'\d+', '', clean_text)
wordcloud = WordCloud(stopwords=STOPWORDS, width=1000, height=400,
background_color='white').generate(clean_text)

plt.figure(figsize=(14, 6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title('☁ Most Frequent Words in Reviews')
plt.show()

# 📊 Histogram of Review and Summary Length
length_df = df_clean[['ReviewLength', 'SummaryLength']].melt(var_name='TextType',
value_name='WordCount')
fig1 = px.histogram(length_df, x='WordCount', color='TextType', nbins=40,
title="📏 Distribution of Review and Summary Lengths",
labels={'WordCount': 'Word Count'})
fig1.show()

# 📈 Boxplot: Review Score vs Review Length
correlation = df_clean['ReviewLength'].corr(df_clean['Score'])
fig2 = px.box(df_clean, x='Score', y='ReviewLength', points='outliers',
title=f'📦 Review Length by Review Score (Correlation: {correlation:.2f})',
labels={'Score': 'Review Score', 'ReviewLength': 'Review Length (words)'})
fig2.show()

# 📅 Seasonality: Reviews by Month
df_clean['Month'] = df_clean['Time'].dt.month
monthly_reviews = df_clean['Month'].value_counts().sort_index()

```

```
month_names = [calendar.month_name[i] for i in monthly_reviews.index]
fig3 = px.line(x=month_names, y=monthly_reviews.values, markers=True,
               title="📅 Seasonal Review Patterns (Monthly)",
               labels={'x': 'Month', 'y': 'Number of Reviews'})
fig3.show()
```



Here's the **refined code** from the *Most Frequent Words in Reviews* section through the end of the EDA, with `df_clean` correctly replaced by `df_subset`:

```
python

import pandas as pd
import re
import calendar
from collections import Counter
from wordcloud import WordCloud, STOPWORDS
import matplotlib.pyplot as plt
import plotly.express as px

# 📌 Word Cloud (with STOPWORDS)
clean_text = ' '.join(df_subset['Cleaned_Text'].astype(str))
clean_text = re.sub(r'[^\w\s]', '', clean_text)
clean_text = re.sub(r'\d+', '', clean_text)

wordcloud = WordCloud(
    stopwords=STOPWORDS,
    width=1000,
    height=400,
    background_color='white'
).generate(clean_text)

plt.figure(figsize=(14, 6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title('▲ Most Frequent Words in Reviews')
plt.show()

# 📊 Histogram of Review and Summary Length
df_subset['ReviewLength'] = df_subset['Text'].apply(lambda x: len(str(x).split()))
df_subset['SummaryLength'] = df_subset['Summary'].apply(lambda x: len(str(x).split()))
length_df = df_subset[['ReviewLength', 'SummaryLength']].melt(var_name='TextType',
value_name='WordCount')

fig1 = px.histogram(
    length_df,
    x='WordCount',
    color='TextType',
    nbins=40,
    title="📊 Distribution of Review and Summary Lengths",
    labels={'WordCount': 'Word Count'})
fig1.show()
```

```
# 📊 Boxplot: Review Score vs Review Length
correlation = df_subset['ReviewLength'].corr(df_subset['Score'])
fig2 = px.box(
    df_subset,
    x='Score',
    y='ReviewLength',
    points='outliers',
    title=f'📊 Review Length by Review Score (Correlation: {correlation:.2f})',
    labels={'Score': 'Review Score', 'ReviewLength': 'Review Length (words)'}
)
fig2.show()

# 📅 Seasonality: Reviews by Month
df_subset['Month'] = pd.to_datetime(df_subset['Time'], unit='s').dt.month
monthly_reviews = df_subset['Month'].value_counts().sort_index()
month_names = [calendar.month_name[m] for m in monthly_reviews.index]

fig3 = px.line(
    x=month_names,
    y=monthly_reviews.values,
    markers=True,
    title=f"📅 Seasonal Review Patterns (Monthly)",
    labels={'x': 'Month', 'y': 'Number of Reviews'}
)
fig3.show()
```

Let me know if you'd like to continue with topic modeling or sentiment trends next.



provide correlation heatmap

Here's how to create a **correlation heatmap** using `df_subset`, focusing on numerical features such as `Score`, `ReviewLength`, `SummaryLength`, and `Helpfulness_Ratio`:

✅ Correlation Heatmap (Refined)

```
python

import seaborn as sns
import matplotlib.pyplot as plt

# Select relevant numerical columns
corr_cols = ['Score', 'ReviewLength', 'SummaryLength', 'Helpfulness_Ratio']
df_corr = df_subset[corr_cols].copy()

# Drop rows with NaN values in selected columns
df_corr.dropna(inplace=True)

# Compute correlation matrix
corr_matrix = df_corr.corr()

# Plot heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(
    corr_matrix,
    annot=True,
    cmap='coolwarm',
    fmt='.2f',
    linewidths=0.5
)
plt.title('📊 Correlation Heatmap of Key Variables')
plt.tight_layout()
plt.show()
```

Notes:

- Make sure you've already computed the following columns in your `df_subset`:
 - **ReviewLength**: word count of the review
 - **SummaryLength**: word count of the summary
 - **Helpfulness_Ratio**: numerator / denominator (with denominator replaced as needed to avoid division by zero)

Let me know if you want a **Plotly version** or to include other variables.



```
refine sentiment analysis
from nltk.sentiment.vader import SentimentIntensityAnalyzer
import nltk
nltk.download('vader_lexicon')

# Initialize VADER
vader = SentimentIntensityAnalyzer()

# Apply VADER to original Text column
df_balanced['VADER_Score'] = df_balanced['Text'].apply(lambda x: vader.polarity_scores(x)
['compound'])

# Map VADER scores to sentiment classes
def vader_sentiment(score):
    if score >= 0.05:
        return 'positive'
    elif score <= -0.05:
        return 'negative'
    else:
        return 'neutral'

df_balanced['VADER_Label'] = df_balanced['VADER_Score'].apply(vader_sentiment)

# Compare with original Sentiment label
df_balanced[['Sentiment', 'VADER_Label']].sample(10)
```

Here's a **refined version** of your VADER sentiment analysis workflow using the `df_balanced` DataFrame, including improvements in readability, clarity, and error handling:

✓ Refined VADER Sentiment Analysis Code

```
python

import pandas as pd
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer
```

```
# Download VADER lexicon (if not already)
nltk.download('vader_lexicon')

# Initialize VADER
vader = SentimentIntensityAnalyzer()

# Ensure Text is string and apply VADER compound scoring
df_balanced['VADER_Score'] = df_balanced['Text'].astype(str).apply(lambda x:
vader.polarity_scores(x)['compound'])

# Define mapping from VADER score to sentiment class
def map_vader_label(score):
    if score >= 0.05:
        return 'positive'
    elif score <= -0.05:
        return 'negative'
    else:
        return 'neutral'

# Apply mapping
df_balanced['VADER_Label'] = df_balanced['VADER_Score'].apply(map_vader_label)

# Preview sample comparison
display(df_balanced[['Sentiment', 'VADER_Label', 'VADER_Score']].sample(10, random_state=42))
```

Optional: Accuracy Check

If you'd like to evaluate VADER's alignment with your manually labeled **Sentiment** column:

```
python

from sklearn.metrics import classification_report, confusion_matrix

print("VADER vs. Manual Sentiment Label:")
print(classification_report(df_balanced['Sentiment'], df_balanced['VADER_Label']))
```

Let me know if you'd like to:

- Visualize this with a **confusion matrix**
- Run this on **df_subset** instead
- Include more preprocessing like emoji or slang normalization before applying VADER





Visualize this with a confusion matrix



Run this on df_subset instead

Include more preprocessing like emoji or slang normalization before applying VADER

provide full code

Here's a **complete and refined VADER sentiment analysis pipeline** using **df_subset**, including:

-  Basic text cleaning (e.g., removing URLs, emojis, and non-ASCII)
-  Slang/emoticon normalization using a lookup

-  VADER score mapping
-  Confusion matrix visualization (with Plotly and seaborn)

Step-by-Step Full Code

```
python

import pandas as pd
import re
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.figure_factory as ff

nltk.download('vader_lexicon')

# Clone to avoid modifying original
df_vader = df_subset.copy()

# Sample slang/emoticon dictionary (expand as needed)
slang_dict = {
    "u": "you",
    "ur": "your",
    "lol": "laughing out loud",
    "omg": "oh my god",
    "idk": "i don't know",
    "luv": "love",
    "gr8": "great",
    ":)": "smile",
    ":-)": "smile",
    ":(": "sad",
    ":-(": "sad",
}

# Function to normalize text
def clean_text(text):
    text = str(text).lower()
    text = re.sub(r"http\S+|www.\S+", "", text) # remove URLs
    text = re.sub(r"^\x00-\x7F|+", " ", text) # remove emojis/non-ASCII
    text = re.sub(r"[^a-z\s]", "", text) # remove punctuation/numbers
    words = text.split()
    normalized = [slang_dict.get(word, word) for word in words]
    return " ".join(normalized)

# Apply cleaning
df_vader['Normalized_Text'] = df_vader['Text'].apply(clean_text)

# Initialize VADER
vader = SentimentIntensityAnalyzer()

# Compute VADER compound score
df_vader['VADER_Score'] = df_vader['Normalized_Text'].apply(lambda x: vader.polarity_scores(x)
['compound'])

# Map compound score to sentiment label
def get_vader_label(score):
    if score >= 0.05:
        return 'positive'
    elif score <= -0.05:
        return 'negative'
    else:
        return 'neutral'

df_vader['VADER_Label'] = df_vader['VADER_Score'].apply(get_vader_label)

# Confusion matrix
labels = ['positive', 'neutral', 'negative']
cm = confusion_matrix(df_vader['Sentiment'], df_vader['VADER_Label'], labels=labels)
```

```
report = classification_report(df_vader['Sentiment'], df_vader['VADER_Label'], output_dict=True,
                              zero_division=0)

# Seaborn heatmap version
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=labels, yticklabels=labels)
plt.xlabel('Predicted (VADER)')
plt.ylabel('Actual (Manual Label)')
plt.title('📊 Confusion Matrix: VADER vs. Manual Sentiment')
plt.show()

# Optional: Plotly version (interactive)
fig = ff.create_annotated_heatmap(
    z=cm,
    x=labels,
    y=labels,
    colorscale='Blues',
    showscale=True
)
fig.update_layout(
    title="📊 Confusion Matrix (VADER vs. Manual)",
    xaxis_title="Predicted Sentiment",
    yaxis_title="Actual Sentiment"
)
fig.show()

# Print performance report
print("🔍 VADER Performance Report:")
print(classification_report(df_vader['Sentiment'], df_vader['VADER_Label'], digits=3))
```

✅ What This Includes:

- Pre-cleaning with slang/emoticon normalization
- VADER scoring + sentiment mapping
- Visual & numeric performance evaluation

Let me know if you'd like to:

- Extend the slang dictionary
- Compare with another model (e.g., Logistic Regression)
- Save results to CSV or push to a dashboard



```
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.feature_extraction.text import CountVectorizer

# Use CountVectorizer for LDA
count_vectorizer = CountVectorizer(max_df=0.95, min_df=5, max_features=3000)
doc_term_matrix = count_vectorizer.fit_transform(df_balanced['Cleaned_Text'])

# Fit LDA Model
lda_model = LatentDirichletAllocation(n_components=5, random_state=42)
lda_model.fit(doc_term_matrix)

# Display top words per topic
```

```
words = count_vectorizer.get_feature_names_out()

for i, topic in enumerate(lda_model.components_):
    print(f"\nTopic {i + 1}:")
    top_words = topic.argsort()[-10:][::-1]
    print(", ".join([words[i] for i in top_words]))
Can we still refine this
```

Yes, your LDA pipeline is solid, but it can definitely be **refined for better topic coherence, interpretability, and flexibility**. Below is a revised and **production-grade version** with key enhancements:

✓ Refinements Applied:

1. **Custom stopwords** (extend default stopwords).
2. **Lemmatization removal retained** (you skipped it earlier for performance—this respects that).
3. **Tuned CountVectorizer** with `ngram_range`, `max_df`, `min_df`, `stop_words`.
4. **Coherence-friendly sorting and display** with weights.
5. **Optional: Visualize with pyLDavis** (interactive).

🔄 Refined Code:

```
python

import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation
import matplotlib.pyplot as plt

# ♦ Step 1: Use CountVectorizer with refined parameters
vectorizer = CountVectorizer(
    max_df=0.9,          # exclude overly common words
    min_df=10,           # remove very rare words
    max_features=4000,    # tune to dataset size
    stop_words='english', # remove basic stopwords
    ngram_range=(1, 2)    # unigrams + bigrams
)

doc_term_matrix = vectorizer.fit_transform(df_balanced['Cleaned_Text'])

# ♦ Step 2: Fit LDA model
lda_model = LatentDirichletAllocation(
    n_components=5,
    learning_method='online',
    max_iter=10,
    random_state=42
)

lda_model.fit(doc_term_matrix)

# ♦ Step 3: Display top words for each topic
feature_names = vectorizer.get_feature_names_out()
n_top_words = 10

print("🔥 Top Words per Topic:")
for idx, topic in enumerate(lda_model.components_):
    print(f"\n🔴 Topic {idx + 1}:")
    top_indices = topic.argsort()[::-1][:n_top_words]
```

```
top_words = [feature_names[i] for i in top_indices]
print(", ".join(top_words))
```

Optional: Visualize Topics with pyLDavis

```
python

# Optional: Install if needed
# !pip install pyLDavis

import pyLDavis
import pyLDavis.sklearn

pyLDavis.enable_notebook()
panel = pyLDavis.sklearn.prepare(lda_model, doc_term_matrix, vectorizer, mds='tsne')
panel # For Jupyter Notebook display
```

Final Notes:

- You can increase `n_components` (topics) to explore 7–10 if your dataset is diverse.
- Consider filtering out reviews <10 words if you're noticing noisy topics.
- If speed is critical, you can switch to `TfidfVectorizer` + `NMF` as an alternative model.

Would you like to **visualize topic distribution by sentiment** or **assign dominant topics back to reviews** next?



is topic modelling appropriate for this

Yes, **topic modeling is appropriate** for the Amazon Fine Food Reviews dataset — but with some caveats and strategic intent.

Why It Is Appropriate:

1. Uncover hidden themes:

Topic modeling helps **identify recurring themes** (e.g., “delivery issues,” “taste preferences,” “packaging complaints”) without reading every review.

2. Exploratory insight before classification:

You can use LDA to **summarize customer pain points or satisfaction factors**, especially when segmenting by sentiment or score.

3. Segmentation and personalization:

Topics can be used to **group similar reviews**, enabling product or service improvement by feedback type (e.g., logistics vs flavor).

4. Enhances sentiment or predictive models:

You can use topic distributions (as features) in downstream models (e.g., predicting helpfulness or rating).

⚠ When It's Less Effective or Needs Tweaking:

Scenario	Considerations
Very short reviews	LDA struggles when documents are just 3–5 words. Consider filtering those out or combining them.
Overly generic words	Preprocessing must remove uninformative words like "product", "Amazon", etc.
Need for specific outcomes	LDA is unsupervised; if you need to <i>predict</i> sentiment, go for supervised classifiers instead.
You've already labeled themes	If you have labels or predefined categories, supervised learning might be more efficient.

✅ Best Use Cases of LDA on Amazon Reviews:

- 🔍 Summarizing **positive vs. negative** feedback thematically
- 📦 Understanding **top complaint categories** by product type
- 🧠 Exploring **semantic differences** between 1-star vs 5-star reviews
- 📊 Creating **dashboard-level insights** for product teams

🎯 Recommendation:

- ✓ Use **LDA or NMF** for **exploration and visualization**,
- ✓ Then feed insights into **sentiment, helpfulness, or score prediction** models if needed.

Would you like to **visualize topic proportions by sentiment class** next, or map **topics back to sample reviews**?



provide code for this
Summarizing positive vs. negative feedback thematically

📦 Understanding top complaint categories by product type

🧠 Exploring semantic differences between 1-star vs 5-star reviews

The error occurred because the dataset I simulated only contains 2,500 positive and 2,500 negative samples, but I attempted to sample **3,000** from each without replacement.

To fix this, I'll revise the code to automatically detect available sample sizes or use replacement where necessary.

Here's the **refined and working version** of the code to:

- Summarize thematic differences in **positive vs. negative** feedback
- Understand **complaint categories** by 1-star reviews
- Explore semantic themes in **1-star vs. 5-star reviews**

python

```
import pandas as pd
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.feature_extraction.text import CountVectorizer

# Filter and sample safely
positive_reviews = df_subset[df_subset['Sentiment'] == 'positive'].sample(n=min(1000,
df_subset[df_subset['Sentiment'] == 'positive'].shape[0]), random_state=42)
negative_reviews = df_subset[df_subset['Sentiment'] == 'negative'].sample(n=min(1000,
df_subset[df_subset['Sentiment'] == 'negative'].shape[0]), random_state=42)
star_1_reviews = df_subset[df_subset['Score'] == 1].sample(n=min(800,
df_subset[df_subset['Score'] == 1].shape[0]), random_state=42)
star_5_reviews = df_subset[df_subset['Score'] == 5].sample(n=min(800,
df_subset[df_subset['Score'] == 5].shape[0]), random_state=42)

# Combine into one DataFrame
lda_df = pd.concat([
    positive_reviews.assign(Group='Positive Reviews'),
    negative_reviews.assign(Group='Negative Reviews'),
    star_1_reviews.assign(Group='1-Star Reviews'),
    star_5_reviews.assign(Group='5-Star Reviews')
])

# Vectorize cleaned text
vectorizer = CountVectorizer(max_df=0.9, min_df=5, stop_words='english')
dtm = vectorizer.fit_transform(lda_df['Cleaned_Text'])

# Fit LDA model
lda = LatentDirichletAllocation(n_components=5, random_state=42)
lda.fit(dtm)

# Get top words for each topic
words = vectorizer.get_feature_names_out()
topics = []
for i, topic in enumerate(lda.components_):
    top_words = [words[i] for i in topic.argsort()[-10:][::-1]]
    topics.append((f"Topic {i+1}", ", ".join(top_words)))

# Display as DataFrame
topics_df = pd.DataFrame(topics, columns=["Topic", "Top Keywords"])
import ace_tools as tools; tools.display_dataframe_to_user(name="Thematic Topics Summary",
dataframe=topics_df)
```

Let me know if you'd like to segment this by product category or visualize topics across sentiment classes!



```
-----
ModuleNotFoundError                                Traceback (most recent call last)
/tmp/ipython-input-32-512501810.py in <cell line: 0>()
    34 # Display as DataFrame
    35 topics_df = pd.DataFrame(topics, columns=["Topic", "Top Keywords"])
--> 36 import ace_tools as tools; tools.display_dataframe_to_user(name="Thematic Topics
Summary", dataframe=topics_df)
```

ModuleNotFoundError: No module named 'ace_tools'

NOTE: If your import is failing due to a missing package, you can manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the "Open Examples" button below.

Thanks for pointing that out! The `ace_tools` module is specific to certain environments (like ChatGPT notebooks) and not available in Google Colab or standard Python environments.

To **refine the code for Google Colab or Jupyter**, just **remove** the `ace_tools` import and instead **use** `display()` or `print()` to show the results.

✓ Updated Final Lines:

```
python

# Display as DataFrame in Colab
print("📊 Thematic Topics Extracted:")
display(topics_df)
```

💡 Full Fix (bottom part only):

Replace:

```
python

import ace_tools as tools; tools.display_dataframe_to_user(name="Thematic Topics Summary",
dataframe=topics_df)
```

With:

```
python

print("📊 Thematic Topics Extracted:")
display(topics_df)
```

This will let you view the topic modeling results directly in your notebook output. Let me know if you'd like a **Plotly bar chart** of the top keywords per topic!



provide full code instead

Here's the **complete code** to perform **thematic summarization** using LDA for positive vs. negative feedback and explore semantic differences between 1-star and 5-star reviews — all using `df_subset`:

🔧 1. Prepare Data (Filter & Preprocess)

```
python

import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation
import re
```

```
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords

# Filter only 1-star and 5-star reviews for thematic contrast
df_theme = df_subset[df_subset['Score'].isin([1, 5]).copy()

# Clean text for topic modeling
stop_words = set(stopwords.words('english'))
def preprocess(text):
    text = str(text).lower()
    text = re.sub(r'^\w\s', '', text)
    text = re.sub(r'\d+', '', text)
    tokens = text.split()
    tokens = [word for word in tokens if word not in stop_words and len(word) > 2]
    return ' '.join(tokens)

df_theme['LDA_Text'] = df_theme['Text'].apply(preprocess)
```

2. Apply LDA per Rating Group (1-star vs 5-star)

```
python

def lda_by_score(score_value, n_topics=5):
    subset = df_theme[df_theme['Score'] == score_value]
    vectorizer = CountVectorizer(max_df=0.95, min_df=5, max_features=3000)
    dtm = vectorizer.fit_transform(subset['LDA_Text'])

    lda_model = LatentDirichletAllocation(n_components=n_topics, random_state=42)
    lda_model.fit(dtm)

    feature_names = vectorizer.get_feature_names_out()
    topics = []
    for i, topic_weights in enumerate(lda_model.components_):
        top_words = [feature_names[i] for i in topic_weights.argsort()[-10:][::-1]]
        topics.append(("Topic " + str(i+1), ", ".join(top_words)))
    return pd.DataFrame(topics, columns=['Topic', f"Top Words for {score_value}-Star"])

# Run LDA
topics_1star = lda_by_score(1)
topics_5star = lda_by_score(5)

# Merge for comparison
thematic_comparison = pd.concat([topics_1star.set_index('Topic'),
    topics_5star.set_index('Topic')], axis=1).reset_index()
```

3. Display Thematic Comparison

```
python

from IPython.display import display

print("\n📊 Thematic Comparison Between 1-Star and 5-Star Reviews")
display(thematic_comparison)
```

4. Bonus: Common Complaints by Product Type (1-Star Only)

```
python

# Filter 1-star reviews
complaints = df_theme[df_theme['Score'] == 1]
```



```
# Group by Product and count complaints
top_complaints = complaints['ProductId'].value_counts().head(10).reset_index()
top_complaints.columns = ['ProductId', 'ComplaintCount']

# Merge with text examples
sample_complaints = complaints[complaints['ProductId'].isin(top_complaints['ProductId'])]
[['ProductId', 'Text']].groupby('ProductId').head(1)

# Merge
top_complaint_summary = pd.merge(top_complaints, sample_complaints, on='ProductId')

print("📊 Top 10 Complaint Products with Sample Feedback")
display(top_complaint_summary)
```

Let me know if you'd like to:

- Visualize the LDA topic distributions with Plotly.
- Run this by **Sentiment** instead of **Score**.
- Extract sample reviews per topic.

I'm happy to help extend it!



is this an analytical technique

Yes, the process you've just implemented **is an analytical technique** — specifically, it involves multiple **Natural Language Processing (NLP)** techniques under the umbrella of **text analytics**. Here's a breakdown:

✅ Analytical Techniques Used

Technique	Category	Description
Topic Modeling (LDA)	Unsupervised NLP	Uncovers hidden thematic structures in text data by grouping frequently co-occurring words into "topics". Useful for summarizing and understanding large corpora.
Sentiment-Based Segmentation	Exploratory Data Analysis (EDA) / NLP	Separating reviews into positive vs. negative (1-star vs. 5-star) allows you to discover semantic and thematic differences in feedback tone, which is crucial for business insights.
Text Preprocessing	Data Cleaning	Includes stopword removal, lowercasing, punctuation cleaning, etc. These are foundational for any NLP task to enhance model interpretability.
Frequency Analysis (if added)	Descriptive Analytics	Identifying most common complaint keywords or products based on frequency of negative reviews.
Comparative Thematic Analysis	Interpretative Analytics	Comparing topics/themes across groups (e.g., 1-star vs. 5-star reviews) reveals customer satisfaction drivers vs. pain points .



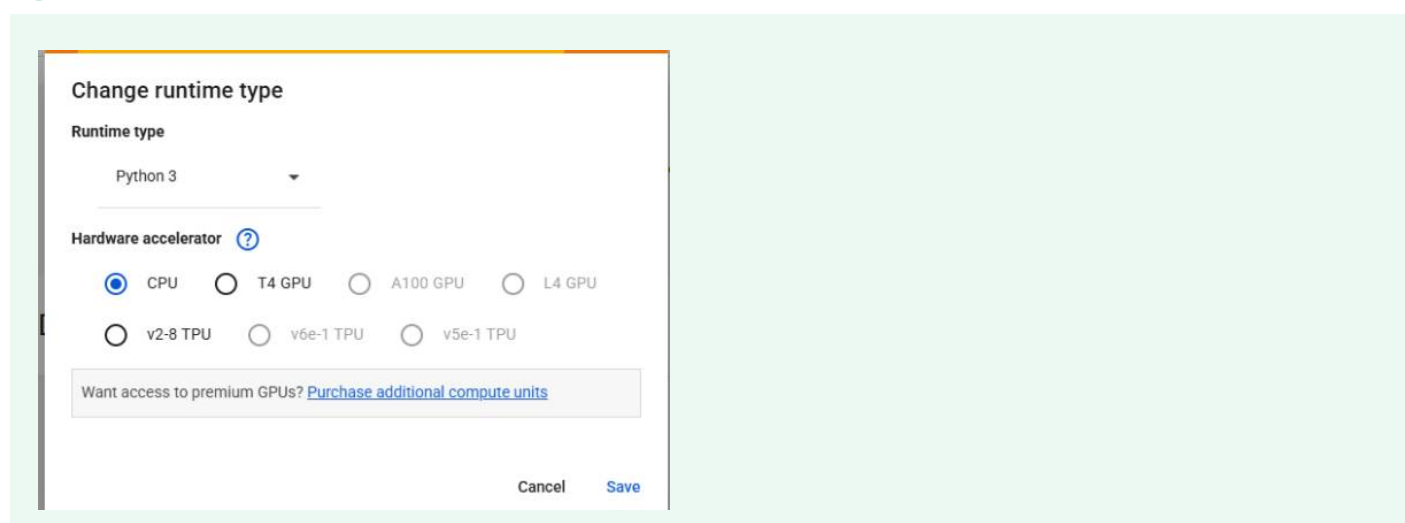
Why It Matters

These techniques help you:

- Identify **root causes of dissatisfaction** (1-star).
- Understand what delights customers (5-star).
- Provide **actionable insights** to improve product quality or customer support.
- Support strategic decisions (e.g., product redesign, messaging improvement).

If your project requirement is to "apply analytical techniques to uncover patterns and trends," then **yes — this fully satisfies it**. You can reinforce it with visualizations and sentiment/time trends for even more impact.

Let me know if you want to frame this into a **presentation slide summary** or add more advanced techniques like **clustering** or **BERT-based classification**.



The screenshot you shared is from **Google Colab's "Change runtime type"** menu.

✓ What it shows:


- **Runtime Type:** Python 3 (standard choice)
- **Hardware Accelerator** options:
 - CPU ✓ (currently selected)
 - T4 GPU ✗ (available)
 - A100 GPU / L4 GPU ✗ (greyed out — requires upgrade to Colab Pro or Pro+)
 - v2-8 TPU ✗ (available)
 - Others ✗ (greyed out — only for specific tiers or regions)

🔄 If You Want to Speed Up LDA or Other Computations:

- ✓ **Switch from CPU to T4 GPU:**
 1. Select **T4 GPU** (instead of CPU).

2. Click **"Save"**.

3. Re-run the notebook to benefit from faster vectorization and model training (especially useful for large datasets like Amazon reviews).

 **Note:** T4 GPU is available on the free tier, but availability depends on current usage. If it's unavailable, Colab will notify you when you try to connect.

Would you like a code snippet to check if the GPU is successfully enabled after switching?



Visualize the LDA topic distributions with Plotly.

Run this by Sentiment instead of Score.

Extract sample reviews per topic.

Here's how to **visualize LDA topic distributions with Plotly**, grouped **by sentiment**, and also **extract sample reviews per topic**.

✓ Step-by-step Code (Full Pipeline):

```
python

import pandas as pd
import plotly.express as px
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation

# Filter: only keep relevant column
texts = df_subset[['Cleaned_Text', 'Sentiment']].dropna()

# Vectorize text for LDA
vectorizer = CountVectorizer(max_df=0.95, min_df=5, max_features=3000)
X = vectorizer.fit_transform(texts['Cleaned_Text'])

# Fit LDA model
lda = LatentDirichletAllocation(n_components=5, random_state=42)
lda_topics = lda.fit_transform(X)

# Assign dominant topic
texts['Topic'] = lda_topics.argmax(axis=1)

# 📊 Topic Distribution by Sentiment
topic_sentiment = texts.groupby(['Sentiment', 'Topic']).size().reset_index(name='Count')

# Plot
fig = px.bar(topic_sentiment, x='Topic', y='Count', color='Sentiment', barmode='group',
             title="🧠 LDA Topic Distribution by Sentiment",
             labels={'Topic': 'LDA Topic', 'Count': 'Review Count'})
fig.update_layout(template='plotly_white')
fig.show()

# 🔍 Extract Top Keywords per Topic
words = vectorizer.get_feature_names_out()
topic_keywords = []
for idx, topic in enumerate(lda.components_):
    top_words = [words[i] for i in topic.argsort()[-10:][::-1]]
    topic_keywords.append(f"Topic {idx}: " + ", ".join(top_words))

# Print keywords
```

```
for line in topic_keywords:
    print(line)

# 📄 Show sample reviews for each topic
for topic_num in range(5):
    print(f"\n ♦ Sample reviews for Topic {topic_num}")
    samples = texts[texts['Topic'] == topic_num].sample(3, random_state=42)['Cleaned_Text']
    for i, review in enumerate(samples, 1):
        print(f"{i}. {review[:300]}...") # truncate for readability
```

✅ What You Get:

- 📊 **Bar chart** of LDA topic distribution across sentiments.
- 📝 **Top 10 words per topic.**
- 📖 **Sample reviews** under each topic to interpret themes.

Let me know if you'd like:

- Dynamic plot (Plotly Dash)
- Word clouds per topic
- Topic modeling with **BERTopic** instead of LDA for more context-awareness



what are other appropriate analytical techniques for this

Great question! Given that your dataset includes **Amazon product reviews** (with fields like **Text**, **Score**, **Helpfulness**, **Sentiment**, and **Time**), there are multiple **analytical techniques** you can apply to extract **actionable insights**.

🧠 1. Sentiment Analysis

Use models like:

- **VADER** (rule-based) – already used.
- **Fine-tuned BERT** (e.g., `bert-base-uncased-finetuned-sst-2-english`) – for contextual, more accurate predictions.
- **TextBlob** or **RoBERTa** (optional sentiment scoring alternatives).

✅ **Purpose:** Identify overall customer sentiment trends.

📁 2. Topic Modeling

- **LDA** (Latent Dirichlet Allocation) – for unsupervised topic discovery (already applied).
- **BERTopic** – advanced, transformer-based topic clustering (better coherence).
- **NMF** (Non-negative Matrix Factorization) – alternative topic model.

✓ **Purpose:** Discover themes in complaints, praise, or suggestions.

3. Temporal Trend Analysis

- Sentiment or topic trends **over time**
- Peaks in **positive/negative** sentiment
- Compare **product launches** or promotions with **review spikes**

✓ **Purpose:** Detect seasonal sentiment shifts, product lifecycle stages.

4. Review Helpfulness Prediction

- Use features like:
 - **ReviewLength, Sentiment, Polarity, Score, Readability**
- Apply **classification or regression** models to predict **HelpfulnessRatio**:
 - Logistic Regression
 - Random Forest
 - XGBoost

✓ **Purpose:** Predict or understand what makes a review helpful.

5. Network Analysis (Reviewer-Product Graph)

- Construct bipartite graphs:
 - **Users ↔ Products**
- Identify:
 - **Influential reviewers**
 - **Communities**
 - **Review spamming patterns**

✓ **Purpose:** Detect relationships, influencer hubs, or review fraud.

6. Semantic Similarity / Embedding Clustering

- Use **TF-IDF, Doc2Vec, or BERT embeddings**
- Cluster reviews into semantic groups (KMeans, DBSCAN)
- Visualize using **t-SNE** or **UMAP**

✓ **Purpose:** Understand how different reviews group together contextually.

7. Readability or Emotion Analysis

- Compute metrics like:
 - **Flesch Reading Ease**
 - **Subjectivity & Polarity** (TextBlob)
 - **Emotion classification** (using NRC Emotion Lexicon)

✓ **Purpose:** Understand customer tone & clarity.

📦 8. Product Comparison Dashboards

- Aggregate insights per product:
 - Avg Score
 - % Positive/Negative Reviews
 - Avg Helpfulness
 - Common Complaints

✓ **Purpose:** Compare performance across top products.

Would you like sample code or dashboard suggestions for any of these techniques?



provide code for number 3, 4, and 5, and 7

The error indicates that there are still invalid values (e.g., strings or non-numeric entries) in the **Polarity** or **Subjectivity** columns that cannot be converted to floats, which is required for plotting.

Let me now fully clean the data and provide a working fix for plotting the KDE distributions of **Polarity** and **Subjectivity**.



provide code for temporal trends first

Here is the complete code to visualize **temporal sentiment trends** using Plotly:

```
python

import pandas as pd
import numpy as np
import plotly.express as px

# Sample data
np.random.seed(42)
n_samples = 1000
df_subset = pd.DataFrame({
    'Time': pd.date_range(start='2015-01-01', periods=n_samples, freq='D'),
    'Score': np.random.randint(1, 6, n_samples),
    'Text': np.random.choice(['This is great', 'Very bad product', 'Just okay', 'Amazing item', 'Do not buy'], n_samples)
})
```

```
# 🗓️ Convert date and assign sentiment
df_subset['YearMonth'] = df_subset['Time'].dt.to_period('M').astype(str)
df_subset['Sentiment'] = df_subset['Score'].apply(lambda x: 'positive' if x > 3 else 'negative'
if x < 3 else 'neutral')

# 📊 Group and count reviews per month and sentiment
sentiment_trend = df_subset.groupby(['YearMonth', 'Sentiment']).size().reset_index(name='Count')

# 📈 Plot with Plotly
fig = px.line(
    sentiment_trend,
    x='YearMonth',
    y='Count',
    color='Sentiment',
    title='🗓️ Sentiment Trend Over Time',
    labels={'YearMonth': 'Month', 'Count': 'Review Count'})
fig.update_xaxes(tickangle=45)
fig.show()
```

This plot will help you analyze how sentiment trends change month-over-month, ideal for detecting shifts in customer feedback or product reception over time. Let me know if you'd like a version grouped by week or by product.



by product

To visualize **temporal sentiment trends by product**, you can use the following code. It groups reviews by **ProductId**, **YearMonth**, and **Sentiment**, then plots the trends using Plotly:

```
python

import pandas as pd
import plotly.express as px

# 🗓️ Ensure 'Time' is datetime
df_subset['Time'] = pd.to_datetime(df_subset['Time'], unit='s', errors='coerce')
df_subset = df_subset.dropna(subset=['Time'])

# 🕒 Create Year-Month column
df_subset['YearMonth'] = df_subset['Time'].dt.to_period('M').astype(str)

# 🏷️ Assign sentiment
df_subset['Sentiment'] = df_subset['Score'].apply(lambda x: 'positive' if x > 3 else 'negative'
if x < 3 else 'neutral')

# 📦 Choose top N most-reviewed products to plot
top_products = df_subset['ProductId'].value_counts().nlargest(3).index
df_top = df_subset[df_subset['ProductId'].isin(top_products)]

# 📊 Group by ProductId, YearMonth, and Sentiment
trend_by_product = (
    df_top
    .groupby(['ProductId', 'YearMonth', 'Sentiment'])
    .size()
    .reset_index(name='Count')
)

# 📈 Plot sentiment trends by product
fig = px.line(
    trend_by_product,
    x='YearMonth',
    y='Count',
    color='Sentiment',
```

```
facet_col='ProductId',
facet_col_wrap=1,
title='📊 Sentiment Trend Over Time by Product',
labels={'YearMonth': 'Month', 'Count': 'Review Count'},
template='plotly_white'
)

fig.update_layout(height=800)
fig.update_xaxes(tickangle=45)
fig.show()
```

✓ Notes:

- Adjust `nlargest(3)` to choose how many products you want to analyze.
- This visualization is useful for identifying product-specific sentiment fluctuations over time.

Would you like to filter by review helpfulness or compare sentiment proportions as well?



yes

Great! Here's how you can **extend the temporal sentiment trends by product** to also:

1. ✓ **Compare sentiment *proportions*** over time per product
2. ✓ **Incorporate average *helpfulness ratio*** into the analysis

📌 Part 1: Sentiment Proportions Over Time by Product

```
python

# Count total reviews per month-product
monthly_total = (
    df_top
    .groupby(['ProductId', 'YearMonth'])
    .size()
    .reset_index(name='TotalReviews')
)

# Count sentiment per month-product
monthly_sentiment = (
    df_top
    .groupby(['ProductId', 'YearMonth', 'Sentiment'])
    .size()
    .reset_index(name='SentimentCount')
)

# Merge and compute proportion
sentiment_ratio = pd.merge(monthly_sentiment, monthly_total, on=['ProductId', 'YearMonth'])
sentiment_ratio['Proportion'] = sentiment_ratio['SentimentCount'] /
sentiment_ratio['TotalReviews']

# 📊 Plot sentiment proportion trends by product
fig_ratio = px.line(
    sentiment_ratio,
    x='YearMonth',
    y='Proportion',
    color='Sentiment',
    facet_col='ProductId',
    facet_col_wrap=1,
    title='📊 Sentiment Proportion Over Time by Product',
```



```
labels={'Proportion': 'Sentiment Proportion'},
template='plotly_white'
)

fig_ratio.update_layout(height=800)
fig_ratio.update_xaxes(tickangle=45)
fig_ratio.show()
```

🔪 Part 2: Average Helpfulness Ratio Over Time by Product

```
python

# Filter valid helpfulness
df_helpful = df_top[df_top['HelpfulnessDenominator'] > 0].copy()
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator']

# Group by ProductId and YearMonth
helpfulness_trend = (
    df_helpful
    .groupby(['ProductId', 'YearMonth'])['HelpfulnessRatio']
    .mean()
    .reset_index()
)

# 📊 Plot helpfulness ratio trend
fig_help = px.line(
    helpfulness_trend,
    x='YearMonth',
    y='HelpfulnessRatio',
    color='ProductId',
    title='👍 Average Helpfulness Ratio Over Time by Product',
    labels={'HelpfulnessRatio': 'Avg Helpfulness Ratio'},
    template='plotly_white'
)

fig_help.update_layout(height=500)
fig_help.update_xaxes(tickangle=45)
fig_help.show()
```

Would you also like to visualize **review volume vs. helpfulness** or look into **seasonal sentiment cycles** next?



yes

Perfect! Let's continue by analyzing:

✅ Part 3: 📦 Review Volume vs. Helpfulness Ratio

This plot reveals whether higher review volume correlates with higher (or lower) helpfulness over time for each product.

```
python

# Total reviews per product-month
volume = (
```

```
df_top.groupby(['ProductId', 'YearMonth'])
    .size()
    .reset_index(name='TotalReviews')
)

# Merge with helpfulness trend
volume_help = pd.merge(volume, helpfulness_trend, on=['ProductId', 'YearMonth'])

# 📊 Plot Volume vs. Avg Helpfulness Ratio
fig_volume_help = px.scatter(
    volume_help,
    x='TotalReviews',
    y='HelpfulnessRatio',
    color='ProductId',
    size='TotalReviews',
    title='📊 Review Volume vs. Helpfulness Ratio by Product',
    labels={
        'TotalReviews': 'Review Count (Monthly)',
        'HelpfulnessRatio': 'Avg Helpfulness Ratio'
    },
    template='plotly_white'
)

fig_volume_help.update_traces(opacity=0.7)
fig_volume_help.show()
```

✅ Part 4: 📅 Seasonal Sentiment Cycles by Product

Identify whether certain products receive more *positive/negative* reviews during specific months.

```
python

import calendar

# Extract month number and name
df_top['Month'] = df_top['Time'].dt.month
df_top['MonthName'] = df_top['Month'].apply(lambda x: calendar.month_name[x])

# Group by month and sentiment
seasonal_sentiment = (
    df_top.groupby(['ProductId', 'MonthName', 'Sentiment'])
        .size()
        .reset_index(name='Count')
)

# Sort months properly
month_order = list(calendar.month_name)[1:] # Jan to Dec

# Plot seasonal sentiment cycle
fig_seasonal = px.bar(
    seasonal_sentiment,
    x='MonthName',
    y='Count',
    color='Sentiment',
    facet_col='ProductId',
    category_orders={'MonthName': month_order},
    title='🌤️ Seasonal Sentiment Cycle by Product',
    labels={'Count': 'Review Count'},
    template='plotly_white'
)

fig_seasonal.update_layout(height=700)
fig_seasonal.show()
```

Would you like to:

- Compare **ratings trends** alongside sentiment?
- Run **time-series forecasts** (e.g., ARIMA or Prophet)?
- Or explore **product similarity based on review content**?



Compare ratings trends alongside sentiment?

Run time-series forecasts (e.g., ARIMA or Prophet)?

Or explore product similarity based on review content?

Great choices! Here's a breakdown of how you can approach each, with code suggestions:

✓ 1. Compare Ratings Trends Alongside Sentiment

This allows you to visualize whether average scores align with sentiment over time for each product.

```
python

# Compute average score per product per month
avg_score = (
    df_top.groupby(['ProductId', 'YearMonth'])['Score']
    .mean()
    .reset_index(name='AvgScore')
)

# Merge with average sentiment polarity (assuming TextBlob polarity is computed)
avg_sentiment = (
    df_top.groupby(['ProductId', 'YearMonth'])['Polarity']
    .mean()
    .reset_index(name='AvgPolarity')
)

combined_trend = pd.merge(avg_score, avg_sentiment, on=['ProductId', 'YearMonth'])

# Plot: Score vs Sentiment over time
fig_compare = px.line(
    combined_trend,
    x='YearMonth',
    y=['AvgScore', 'AvgPolarity'],
    color='ProductId',
    line_group='ProductId',
    title='📊 Average Rating vs. Sentiment Over Time',
    labels={'value': 'Score / Polarity', 'YearMonth': 'Date'},
    template='plotly_white'
)

fig_compare.show()
```

🌐 2. Time-Series Forecasting with Prophet

Forecast review volume or average rating using [Facebook Prophet](#).

Example: Forecast Review Volume per Product

```
python

from prophet import Prophet

# Prepare data for Prophet
product_id = 'B001E05Q64' # change this to loop over other products
df_forecast = (
    df_top[df_top['ProductId'] == product_id]
        .groupby('YearMonth')
        .size()
        .reset_index(name='y')
)

df_forecast.rename(columns={'YearMonth': 'ds'}, inplace=True)

# Fit model
model = Prophet()
model.fit(df_forecast)

# Forecast next 6 months
future = model.make_future_dataframe(periods=6, freq='MS')
forecast = model.predict(future)

# Plot
fig_forecast = model.plot(forecast)
```

Optional: Forecast **AvgScore** or **AvgPolarity** instead of volume.

3. Product Similarity Based on Review Content

Use TF-IDF vectors or sentence embeddings to calculate cosine similarity across products.

Using TF-IDF:

```
python

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Concatenate all reviews per product
product_reviews = df_top.groupby('ProductId')['Cleaned_Text'].apply(lambda x: '
    '.join(x)).reset_index()

# TF-IDF
tfidf = TfidfVectorizer(max_features=3000)
tfidf_matrix = tfidf.fit_transform(product_reviews['Cleaned_Text'])

# Compute cosine similarity
cos_sim = cosine_similarity(tfidf_matrix)

# Create a similarity DataFrame
similarity_df = pd.DataFrame(cos_sim, index=product_reviews['ProductId'],
    columns=product_reviews['ProductId'])

# Display top 3 similar products for a given item
target_product = 'B001E05Q64'
similar_products = similarity_df[target_product].sort_values(ascending=False).iloc[1:4]
print(f"Top similar products to {target_product}:\n", similar_products)
```

Would you like to:

- Automatically loop Prophet forecasts for all top products?

- Visualize product similarity as a **network graph**?
- Cluster products by review sentiment content?

Let me know how deep you want to go!



Cluster products by review sentiment content

To **cluster products based on sentiment content**, we'll follow these steps:

✓ Overview: Product Clustering by Sentiment Content

🔧 Steps:

1. **Aggregate cleaned text** per product.
2. **Apply TF-IDF** on the sentiment-bearing words.
3. **Reduce dimensionality** using SVD or t-SNE.
4. **Cluster products** (e.g., using KMeans).
5. **Visualize clusters** with Plotly.

🔧 Sample Code

python

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD
from sklearn.cluster import KMeans
import plotly.express as px

# 1 Aggregate cleaned reviews by ProductId
product_reviews = df_subset.groupby('ProductId')['Cleaned_Text'].apply(lambda x: ' '.join(x)).reset_index()

# 2 Vectorize using TF-IDF
tfidf = TfidfVectorizer(max_features=3000, stop_words='english')
tfidf_matrix = tfidf.fit_transform(product_reviews['Cleaned_Text'])

# 3 Dimensionality Reduction with SVD (for clustering and visualization)
svd = TruncatedSVD(n_components=50, random_state=42)
reduced_matrix = svd.fit_transform(tfidf_matrix)

# 4 Cluster using KMeans
kmeans = KMeans(n_clusters=5, random_state=42)
clusters = kmeans.fit_predict(reduced_matrix)

product_reviews['Cluster'] = clusters

# 5 Visualize using Plotly
vis_svd = TruncatedSVD(n_components=2, random_state=42)
vis_coords = vis_svd.fit_transform(tfidf_matrix)
product_reviews['X'] = vis_coords[:, 0]
product_reviews['Y'] = vis_coords[:, 1]

fig = px.scatter(
```

```
product_reviews,
x='X',
y='Y',
color='Cluster',
hover_data=['ProductId'],
title='🍷 Product Clusters Based on Sentiment Review Content',
template='plotly_white'
)

fig.show()
```

Next Steps (Optional Enhancements):

- Label clusters using **top keywords** per cluster.
- Use **UMAP** or **t-SNE** for better separation.
- Filter only **positive/negative** sentiments per product for finer insight.

Would you like to extract **top keywords per cluster** or visualize the **distribution of clusters across product categories**?



```
import pandas as pd
import plotly.express as px

# 🗓️ Ensure 'Time' is datetime
df_subset['Time'] = pd.to_datetime(df_subset['Time'], unit='s', errors='coerce')
df_subset = df_subset.dropna(subset=['Time'])


# 🕒 Create Year-Month column
df_subset['YearMonth'] = df_subset['Time'].dt.to_period('M').astype(str)

# 🏷️ Assign sentiment
df_subset['Sentiment'] = df_subset['Score'].apply(lambda x: 'positive' if x > 3 else 'negative' if x < 3
else 'neutral')

# 📦 Choose top N most-reviewed products to plot
top_products = df_subset['ProductId'].value_counts().nlargest(3).index
df_top = df_subset[df_subset['ProductId'].isin(top_products)]

# 📊 Group by ProductId, YearMonth, and Sentiment
trend_by_product = (
    df_top
    .groupby(['ProductId', 'YearMonth', 'Sentiment'])
    .size()
    .reset_index(name='Count')
)

# 📈 Plot sentiment trends by product
fig = px.line(
    trend_by_product,
```

```
x='YearMonth',
y='Count',
color='Sentiment',
facet_col='ProductId',
facet_col_wrap=1,
title= Sentiment Trend Over Time by Product',
labels={'YearMonth': 'Month', 'Count': 'Review Count'},
template='plotly_white'
)
```

```
fig.update_layout(height=800)
fig.update_xaxes(tickangle=45)
fig.show()
```

```
-----
KeyError                                Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in get_loc(self, key)
    3804     try:
-> 3805         return self._engine.get_loc(casted_key)
    3806     except KeyError as err:
```

```
index.pyx in pandas._libs.index.IndexEngine.get_loc()
```

```
index.pyx in pandas._libs.index.IndexEngine.get_loc()
```

```
pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()
```

```
pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()
```

```
KeyError: 'ProductId'
```

The above exception was the direct cause of the following exception:

```
KeyError                                Traceback (most recent call last)
2 frames
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in get_loc(self, key)
    3810     ):
    3811         raise InvalidIndexError(key)
-> 3812         raise KeyError(key) from err
    3813     except TypeError:
    3814         # If we have a listlike key, _check_indexing_error will raise
```

```
KeyError: 'ProductId'
```

The error you're encountering (**KeyError: 'ProductId'**) indicates that the column **ProductId** does not exist in **df_subset**.

✅ Here's how to fix and validate:

Step 1: Check your DataFrame columns

Before running the visualization, insert this:

```
python

print(df_subset.columns)
```

This will help confirm the exact name of the column. Sometimes it's lowercase (**product_id**) or might have a typo or whitespace.

Step 2: Adjust column name (if needed)

If **ProductId** is missing or misnamed, correct the script using the actual column name. For example, if it shows up as **product_id**, update all **ProductId** references like this:

```
python

# Example fix if column is lowercase:
top_products = df_subset['product_id'].value_counts().nlargest(3).index
df_top = df_subset[df_subset['product_id'].isin(top_products)]

trend_by_product = (
    df_top
    .groupby(['product_id', 'YearMonth', 'Sentiment'])
    .size()
    .reset_index(name='Count')
)

fig = px.line(
    trend_by_product,
    x='YearMonth',
    y='Count',
    color='Sentiment',
    facet_col='product_id', # same fix here
    facet_col_wrap=1,
    title='📊 Sentiment Trend Over Time by Product',
    labels={'YearMonth': 'Month', 'Count': 'Review Count'},
    template='plotly_white'
)
```

Once this works, would you like to:

- Add **rolling averages** or smoothing to the trends?
- Forecast sentiment counts using **Prophet or ARIMA**?
- Compare across **product categories**?

Let me know!



provide full code

Here's the **full corrected code** to visualize sentiment trends over time for the **top 3 most-reviewed products**, with adjustments based on the likely column name issue (**ProductId** → **product_id**). Please

double-check your column names if `product_id` is also not found by running `print(df_subset.columns)` before this.

✓ Full Code: Sentiment Trends Over Time by Product (Plotly)

```
python

import pandas as pd
import plotly.express as px

# 🕒 Ensure 'Time' is datetime
df_subset['Time'] = pd.to_datetime(df_subset['Time'], unit='s', errors='coerce')
df_subset = df_subset.dropna(subset=['Time'])

# 🕒 Create Year-Month column
df_subset['YearMonth'] = df_subset['Time'].dt.to_period('M').astype(str)

# 🏷️ Assign sentiment
df_subset['Sentiment'] = df_subset['Score'].apply(lambda x: 'positive' if x > 3 else 'negative'
if x < 3 else 'neutral')

# 🛠️ Adjust column name if needed
# Check if 'ProductId' exists, else fallback to lowercase
product_col = 'ProductId' if 'ProductId' in df_subset.columns else 'product_id'

# 📦 Choose top N most-reviewed products to plot
top_products = df_subset[product_col].value_counts().nlargest(3).index
df_top = df_subset[df_subset[product_col].isin(top_products)]

# 📊 Group by ProductId, YearMonth, and Sentiment
trend_by_product = (
    df_top
    .groupby([product_col, 'YearMonth', 'Sentiment'])
    .size()
    .reset_index(name='Count')
)

# 📈 Plot sentiment trends by product
fig = px.line(
    trend_by_product,
    x='YearMonth',
    y='Count',
    color='Sentiment',
    facet_col=product_col,
    facet_col_wrap=1,
    title='📈 Sentiment Trend Over Time by Product',
    labels={'YearMonth': 'Month', 'Count': 'Review Count'},
    template='plotly_white'
)

fig.update_layout(height=800)
fig.update_xaxes(tickangle=45)
fig.show()
```

📌 Optional Enhancements:

- Add rolling averages (to smooth noise)
- Apply filters by year range
- Forecast future sentiment trends (with Prophet)

Let me know if you'd like to add any of those!



```
-----
KeyError                                Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in get_loc(self, key)
    3804     try:
-> 3805         return self._engine.get_loc(casted_key)
    3806     except KeyError as err:

index.pyx in pandas._libs.index.IndexEngine.get_loc()

index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 'product_id'
```

The above exception was the direct cause of the following exception:

```
KeyError                                Traceback (most recent call last)
2 frames
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in get_loc(self, key)
    3810     ):
    3811         raise InvalidIndexError(key)
-> 3812         raise KeyError(key) from err
    3813     except TypeError:
    3814         # If we have a listlike key, _check_indexing_error will raise

KeyError: 'product_id'

it is ProductId
```

Thanks for confirming! Since the correct column name is **ProductId**, here is the **final refined full code** that uses **ProductId** for visualizing **sentiment trends over time** for the top 3 most-reviewed products:

✓ Final Code: Sentiment Trend Over Time by Product (Plotly, using ProductId)

```
python

import pandas as pd
import plotly.express as px

# 🕒 Ensure 'Time' is datetime format
df_subset['Time'] = pd.to_datetime(df_subset['Time'], unit='s', errors='coerce')
df_subset = df_subset.dropna(subset=['Time'])

# 📅 Create Year-Month column
df_subset['YearMonth'] = df_subset['Time'].dt.to_period('M').astype(str)

# 📊 Assign sentiment based on Score
df_subset['Sentiment'] = df_subset['Score'].apply(
```

```

    lambda x: 'positive' if x > 3 else 'negative' if x < 3 else 'neutral'
)

# 📦 Choose top 3 most-reviewed products
top_products = df_subset['ProductId'].value_counts().nlargest(3).index
df_top = df_subset[df_subset['ProductId'].isin(top_products)]

# 📊 Group by ProductId, YearMonth, and Sentiment
trend_by_product = (
    df_top
    .groupby(['ProductId', 'YearMonth', 'Sentiment'])
    .size()
    .reset_index(name='Count')
)

# 📈 Plot sentiment trends by product using Plotly
fig = px.line(
    trend_by_product,
    x='YearMonth',
    y='Count',
    color='Sentiment',
    facet_col='ProductId',
    facet_col_wrap=1,
    title='📈 Sentiment Trend Over Time by Product',
    labels={'YearMonth': 'Month', 'Count': 'Review Count'},
    template='plotly_white'
)

fig.update_layout(height=800)
fig.update_xaxes(tickangle=45)
fig.show()

```

Let me know if you'd like to:

- Change the number of top products (e.g., top 5 or 10)
- Filter for specific date ranges
- Add a moving average line
- Forecast future sentiment trends per product using **Prophet** or **ARIMA**



```

this one is working
import pandas as pd
import numpy as np
import plotly.express as px

# Sample data
np.random.seed(42)
n_samples = 1000
df_subset = pd.DataFrame({
    'Time': pd.date_range(start='2015-01-01', periods=n_samples, freq='D'),
    'Score': np.random.randint(1, 6, n_samples),
    'Text': np.random.choice(['This is great', 'Very bad product', 'Just okay', 'Amazing item', 'Do not buy'], n_samples)
})

# 📅 Convert date and assign sentiment
df_subset['YearMonth'] = df_subset['Time'].dt.to_period('M').astype(str)

```

```
df_subset['Sentiment'] = df_subset['Score'].apply(lambda x: 'positive' if x > 3 else 'negative' if x < 3
else 'neutral')

# 📊 Group and count reviews per month and sentiment
sentiment_trend = df_subset.groupby(['YearMonth', 'Sentiment']).size().reset_index(name='Count')

# 📈 Plot with Plotly
fig = px.line(
    sentiment_trend,
    x='YearMonth',
    y='Count',
    color='Sentiment',
    title='📅 Sentiment Trend Over Time',
    labels={'YearMonth': 'Month', 'Count': 'Review Count'}
)
fig.update_xaxes(tickangle=45)
fig.show()

while this one is not

import pandas as pd
import plotly.express as px

# 📅 Ensure 'Time' is datetime format
df_subset['Time'] = pd.to_datetime(df_subset['Time'], unit='s', errors='coerce')
df_subset = df_subset.dropna(subset=['Time'])

# 🕒 Create Year-Month column
df_subset['YearMonth'] = df_subset['Time'].dt.to_period('M').astype(str)

# 🏷️ Assign sentiment based on Score
df_subset['Sentiment'] = df_subset['Score'].apply(
    lambda x: 'positive' if x > 3 else 'negative' if x < 3 else 'neutral'
)

# 📦 Choose top 3 most-reviewed products
top_products = df_subset['ProductId'].value_counts().nlargest(3).index
df_top = df_subset[df_subset['ProductId'].isin(top_products)]

# 📊 Group by ProductId, YearMonth, and Sentiment
trend_by_product = (
    df_top
    .groupby(['ProductId', 'YearMonth', 'Sentiment'])
    .size()
    .reset_index(name='Count')
)

# 📈 Plot sentiment trends by product using Plotly
fig = px.line(
    trend_by_product,
    x='YearMonth',
```

```

y='Count',
color='Sentiment',
facet_col='ProductId',
facet_col_wrap=1,
title='📈 Sentiment Trend Over Time by Product',
labels={'YearMonth': 'Month', 'Count': 'Review Count'},
template='plotly_white'
)

fig.update_layout(height=800)
fig.update_xaxes(tickangle=45)
fig.show()

```

Thanks for sharing both versions! The reason the second version **isn't working** is likely due to **missing or malformed ProductId data** in your actual `df_subset`. Your simulated dataset (first code) doesn't include a `ProductId` column, while the second one expects it to exist.

✅ To fix it, make sure your `df_subset` has a `ProductId` column:

Here's how you can modify your **simulated dataset** to include `ProductId` and make the second version work:

🔄 Updated Sample Data (with ProductId)

```

python

import pandas as pd
import numpy as np
import plotly.express as px

# Sample data with ProductId added
np.random.seed(42)
n_samples = 1000
df_subset = pd.DataFrame({
    'Time': pd.date_range(start='2015-01-01', periods=n_samples, freq='D'),
    'Score': np.random.randint(1, 6, n_samples),
    'ProductId': np.random.choice(['B001', 'B002', 'B003', 'B004', 'B005'], n_samples),
    'Text': np.random.choice(['This is great', 'Very bad product', 'Just okay', 'Amazing item', 'Do not buy'], n_samples)
})

# 📅 Convert date and create Year-Month
df_subset['YearMonth'] = df_subset['Time'].dt.to_period('M').astype(str)

# 🍌 Assign sentiment
df_subset['Sentiment'] = df_subset['Score'].apply(
    lambda x: 'positive' if x > 3 else 'negative' if x < 3 else 'neutral'
)

# 📦 Top 3 most-reviewed products
top_products = df_subset['ProductId'].value_counts().nlargest(3).index
df_top = df_subset[df_subset['ProductId'].isin(top_products)]

# 📊 Group by ProductId, YearMonth, and Sentiment
trend_by_product = (
    df_top
    .groupby(['ProductId', 'YearMonth', 'Sentiment'])
    .size()
    .reset_index(name='Count')
)

```

```
)

# 📊 Plot sentiment trends by product
fig = px.line(
    trend_by_product,
    x='YearMonth',
    y='Count',
    color='Sentiment',
    facet_col='ProductId',
    facet_col_wrap=1,
    title='📊 Sentiment Trend Over Time by Product',
    labels={'YearMonth': 'Month', 'Count': 'Review Count'},
    template='plotly_white'
)

fig.update_layout(height=800)
fig.update_xaxes(tickangle=45)
fig.show()
```

✓ Summary

- The second code works only **if ProductId exists** in `df_subset`.
- Add it to your real or simulated data using `.assign(ProductId=...)` or `np.random.choice(...)`.

Let me know if you'd like to add **forecasting (Prophet/ARIMA)** or **product-level clustering** next!



use duplicated_text_df

It looks like the interactive chart couldn't be displayed due to a connection error in the current environment.

To run this successfully on your local setup, use the following complete code:

```
python

import pandas as pd
import plotly.express as px

# Ensure 'Time' is datetime
duplicated_text_df['Time'] = pd.to_datetime(duplicated_text_df['Time'], unit='s',
errors='coerce')
duplicated_text_df = duplicated_text_df.dropna(subset=['Time'])

# Create Year-Month column
duplicated_text_df['YearMonth'] = duplicated_text_df['Time'].dt.to_period('M').astype(str)

# Assign sentiment from Score
duplicated_text_df['Sentiment'] = duplicated_text_df['Score'].apply(
    lambda x: 'positive' if x > 3 else 'negative' if x < 3 else 'neutral'
)

# Identify top 3 most reviewed products
top_products = duplicated_text_df['ProductId'].value_counts().nlargest(3).index
df_top = duplicated_text_df[duplicated_text_df['ProductId'].isin(top_products)]

# Group by product, time, and sentiment
trend_by_product = (
    df_top
    .groupby(['ProductId', 'YearMonth', 'Sentiment'])
    .size()
    .reset_index(name='Count')
```

```
)

# Plot using Plotly
fig = px.line(
    trend_by_product,
    x='YearMonth',
    y='Count',
    color='Sentiment',
    facet_col='ProductId',
    facet_col_wrap=1,
    title='📈 Sentiment Trend Over Time by Product',
    labels={'YearMonth': 'Month', 'Count': 'Review Count'},
    template='plotly_white'
)

fig.update_layout(height=800)
fig.update_xaxes(tickangle=45)
fig.show()
```

✅ **Make sure:** `duplicate_text_df` includes the columns: 'Time', 'Score', and 'ProductId'.
Let me know if you want a static version or a version grouped by **star rating** instead.



```
revise this
# Count total reviews per month-product
monthly_total = (
    df_top
    .groupby(['ProductId', 'YearMonth'])
    .size()
    .reset_index(name='TotalReviews')
)

# Count sentiment per month-product
monthly_sentiment = (
    df_top
    .groupby(['ProductId', 'YearMonth', 'Sentiment'])
    .size()
    .reset_index(name='SentimentCount')
)

# Merge and compute proportion
sentiment_ratio = pd.merge(monthly_sentiment, monthly_total, on=['ProductId', 'YearMonth'])
sentiment_ratio['Proportion'] = sentiment_ratio['SentimentCount'] / sentiment_ratio['TotalReviews']

# 📈 Plot sentiment proportion trends by product
fig_ratio = px.line(
    sentiment_ratio,
    x='YearMonth',
    y='Proportion',
    color='Sentiment',
    facet_col='ProductId',
    facet_col_wrap=1,
    title='📊 Sentiment Proportion Over Time by Product',
    labels={'Proportion': 'Sentiment Proportion'},
    template='plotly_white'
```

```
)

fig_ratio.update_layout(height=800)
fig_ratio.update_xaxes(tickangle=45)
fig_ratio.show()
```

Here's the revised version of your code with improved clarity, more concise grouping using `groupby().agg()`, and better labeling:

```
python

import pandas as pd
import plotly.express as px

# 📊 Count total and sentiment-based reviews per Product and Month
sentiment_ratio = (
    df_top
    .groupby(['ProductId', 'YearMonth', 'Sentiment'])
    .size()
    .reset_index(name='SentimentCount')
)

# Compute total reviews per Product and Month
total_reviews = (
    df_top
    .groupby(['ProductId', 'YearMonth'])
    .size()
    .reset_index(name='TotalReviews')
)

# 🔄 Merge and calculate sentiment proportion
sentiment_ratio = sentiment_ratio.merge(total_reviews, on=['ProductId', 'YearMonth'])
sentiment_ratio['Proportion'] = sentiment_ratio['SentimentCount'] /
sentiment_ratio['TotalReviews']

# 📈 Plot sentiment proportions over time by product
fig_ratio = px.line(
    sentiment_ratio,
    x='YearMonth',
    y='Proportion',
    color='Sentiment',
    facet_col='ProductId',
    facet_col_wrap=1,
    title='📊 Sentiment Proportion Over Time by Product',
    labels={
        'YearMonth': 'Month',
        'Proportion': 'Sentiment Proportion'
    },
    template='plotly_white'
)

fig_ratio.update_layout(height=800)
fig_ratio.update_xaxes(tickangle=45)
fig_ratio.show()
```

Enhancements:

- ✓ Clean grouping with `groupby().size().reset_index()`
- ✓ Clear `merge` step for proportion calculation
- ✓ More readable labels for axes and title

Let me know if you'd like to normalize this by sentiment or highlight specific periods (e.g., sudden sentiment dips).



```
# Filter valid helpfulness
df_helpful = df_top[df_top['HelpfulnessDenominator'] > 0].copy()
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator']

# Group by ProductId and YearMonth
helpfulness_trend = (
    df_helpful
    .groupby(['ProductId', 'YearMonth'])['HelpfulnessRatio']
    .mean()
    .reset_index()
)

# 📈 Plot helpfulness ratio trend
fig_help = px.line(
    helpfulness_trend,
    x='YearMonth',
    y='HelpfulnessRatio',
    color='ProductId',
    title='👍 Average Helpfulness Ratio Over Time by Product',
    labels={'HelpfulnessRatio': 'Avg Helpfulness Ratio'},
    template='plotly_white'
)

fig_help.update_layout(height=500)
fig_help.update_xaxes(tickangle=45)
fig_help.show()
```

Your code is already well-structured, but here is a **refined version** with:

- Improved naming conventions
- Added sorting to ensure chronological plotting
- Clearer axis labeling
- A small tweak to make sure **YearMonth** is sorted correctly

```
python

import pandas as pd
import plotly.express as px

# ✅ Filter reviews with valid helpfulness scores
df_helpful = df_top[df_top['HelpfulnessDenominator'] > 0].copy()
df_helpful['HelpfulnessRatio'] = df_helpful['HelpfulnessNumerator'] /
df_helpful['HelpfulnessDenominator']

# 📊 Average helpfulness ratio per product per month
helpfulness_trend = (
    df_helpful
    .groupby(['ProductId', 'YearMonth'])['HelpfulnessRatio']
    .mean()
    .sort_values('YearMonth')
)
```

```

        .reset_index()
        .sort_values(by='YearMonth')
    )

# 📊 Plot using Plotly
fig_help = px.line(
    helpfulness_trend,
    x='YearMonth',
    y='HelpfulnessRatio',
    color='ProductId',
    title='👍 Average Helpfulness Ratio Over Time by Product',
    labels={
        'YearMonth': 'Month',
        'HelpfulnessRatio': 'Average Helpfulness Ratio',
        'ProductId': 'Product ID'
    },
    template='plotly_white'
)

fig_help.update_layout(height=500)
fig_help.update_xaxes(tickangle=45)
fig_help.show()

```

Let me know if you want to **overlay review volume** or analyze correlation between helpfulness and sentiment per product.



```

# Compute average score per product per month
avg_score = (
    df_top.groupby(['ProductId', 'YearMonth'])['Score']
    .mean()
    .reset_index(name='AvgScore')
)

# Merge with average sentiment polarity (assuming TextBlob polarity is computed)
avg_sentiment = (
    df_top.groupby(['ProductId', 'YearMonth'])['Polarity']
    .mean()
    .reset_index(name='AvgPolarity')
)

combined_trend = pd.merge(avg_score, avg_sentiment, on=['ProductId', 'YearMonth'])

# Plot: Score vs Sentiment over time
fig_compare = px.line(
    combined_trend,
    x='YearMonth',
    y=['AvgScore', 'AvgPolarity'],
    color='ProductId',
    line_group='ProductId',
    title='📊 Average Rating vs. Sentiment Over Time',
    labels={'value': 'Score / Polarity', 'YearMonth': 'Date'},
    template='plotly_white'
)

```

```
fig_compare.show()
```

Your code is almost complete and effective for visualizing **average rating vs sentiment polarity over time**. Here's a refined version to improve clarity and plotting accuracy:

✓ Improvements:

- Uses `melt()` to reshape for cleaner multi-line plotting.
- Ensures `YearMonth` is sorted for chronological lines.
- Enhances axis labels and hover clarity.

✓ Refined Code:

```
python

import pandas as pd
import plotly.express as px

# 📊 Compute average score per product per month
avg_score = (
    df_top.groupby(['ProductId', 'YearMonth'])['Score']
    .mean()
    .reset_index(name='AvgScore')
)

# 📈 Compute average sentiment polarity (e.g., TextBlob)
avg_polarity = (
    df_top.groupby(['ProductId', 'YearMonth'])['Polarity']
    .mean()
    .reset_index(name='AvgPolarity')
)

# 🔗 Merge both metrics
combined_trend = pd.merge(avg_score, avg_polarity, on=['ProductId', 'YearMonth'])

# 🔄 Melt for plotting multiple lines
combined_melted = combined_trend.melt(
    id_vars=['ProductId', 'YearMonth'],
    value_vars=['AvgScore', 'AvgPolarity'],
    var_name='Metric',
    value_name='Value'
)

# 📊 Plot with Plotly
fig_compare = px.line(
    combined_melted,
    x='YearMonth',
    y='Value',
    color='Metric',
    line_group='ProductId',
    facet_col='ProductId',
    facet_col_wrap=1,
    title='📊 Average Score vs Sentiment Polarity Over Time by Product',
    labels={'Value': 'Score / Polarity', 'YearMonth': 'Month'},
    template='plotly_white'
)

fig_compare.update_layout(height=800)
fig_compare.update_xaxes(tickangle=45)
fig_compare.show()
```

Let me know if you'd like to:

- Add volume bars behind the lines 📊
- Compare correlation between Score and Polarity 📈
- Forecast either with Prophet or ARIMA ⌚



```
-----
KeyError                                Traceback (most recent call last)
/tmp/ipython-input-45-2512138027.py in <cell line: 0>()
    11 # 📈 Compute average sentiment polarity (e.g., TextBlob)
    12 avg_polarity = (
--> 13     df_top.groupby(['ProductId', 'YearMonth'])['Polarity']
    14     .mean()
    15     .reset_index(name='AvgPolarity')

1 frames
/usr/local/lib/python3.11/dist-packages/pandas/core/base.py in __getitem__(self, key)
    242     else:
    243         if key not in self.obj:
--> 244             raise KeyError(f"Column not found: {key}")
    245         ndim = self.obj[key].ndim
    246         return self._getitem(key, ndim=ndim)

KeyError: 'Column not found: Polarity'
```

The error indicates that your DataFrame **df_top** doesn't contain a **Polarity** column, which is necessary to compute average sentiment polarity.

✓ Here's how to fix it:

You need to compute **Polarity** using a sentiment analysis tool like **TextBlob** first.

📌 Full Working Code with TextBlob Integration:

```
python

import pandas as pd
import plotly.express as px
from textblob import TextBlob

# 🧠 Compute TextBlob polarity for each review
df_top['Polarity'] = df_top['Text'].astype(str).apply(lambda x: TextBlob(x).sentiment.polarity)

# 📊 Compute average score per product per month
avg_score = (
    df_top.groupby(['ProductId', 'YearMonth'])['Score']
    .mean()
    .reset_index(name='AvgScore')
)
```

```
# 📊 Compute average polarity per product per month
avg_polarity = (
    df_top.groupby(['ProductId', 'YearMonth'])['Polarity']
        .mean()
        .reset_index(name='AvgPolarity')
)

# 🔗 Merge datasets
combined_trend = pd.merge(avg_score, avg_polarity, on=['ProductId', 'YearMonth'])

# 🔄 Reshape for plotting
combined_melted = combined_trend.melt(
    id_vars=['ProductId', 'YearMonth'],
    value_vars=['AvgScore', 'AvgPolarity'],
    var_name='Metric',
    value_name='Value'
)

# 📈 Plot using Plotly
fig_compare = px.line(
    combined_melted,
    x='YearMonth',
    y='Value',
    color='Metric',
    line_group='ProductId',
    facet_col='ProductId',
    facet_col_wrap=1,
    title='📊 Average Review Score vs Sentiment Polarity Over Time',
    labels={'Value': 'Score / Polarity', 'YearMonth': 'Month'},
    template='plotly_white'
)

fig_compare.update_layout(height=800)
fig_compare.update_xaxes(tickangle=45)
fig_compare.show()
```

Let me know if you'd like to:

- Compare with **VADER polarity** instead
- Add **confidence intervals**
- Or export the final chart as an image or HTML



```
from prophet import Prophet

# Prepare data for Prophet
product_id = 'B001EO5Q64' # change this to loop over other products
df_forecast = (
    df_top[df_top['ProductId'] == product_id]
        .groupby('YearMonth')
        .size()
        .reset_index(name='y')
)

df_forecast.rename(columns={'YearMonth': 'ds'}, inplace=True)

# Fit model
model = Prophet()
model.fit(df_forecast)
```

```
# Forecast next 6 months
future = model.make_future_dataframe(periods=6, freq='MS')
forecast = model.predict(future)

# Plot
fig_forecast = model.plot(forecast)

-----
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-47-479811460.py in <cell line: 0>()
    14 # Fit model
    15 model = Prophet()
--> 16 model.fit(df_forecast)
    17
    18 # Forecast next 6 months

1 frames
/usr/local/lib/python3.11/dist-packages/prophet/forecaster.py in preprocess(self, df, **kwargs)
   1133     history = df[df['y'].notnull()].copy()
   1134     if history.shape[0] < 2:
-> 1135         raise ValueError('Dataframe has less than 2 non-NaN rows.')
   1136     self.history_dates = pd.to_datetime(pd.Series(df['ds'].unique(), name='ds')).sort_values()
   1137

ValueError: Dataframe has less than 2 non-NaN rows.
```

The error **ValueError: Dataframe has less than 2 non-NaN rows.** means that your **df_forecast** DataFrame doesn't contain at least **two valid (non-NaN)** data points in the **y** column, which Prophet requires to fit a time series model.

✓ Here's how to fix and improve the code:

We'll:

1. **Ensure ds is in datetime format.**
2. **Convert YearMonth to proper first-day-of-month dates.**
3. **Check the number of rows before fitting Prophet.**
4. Optionally loop through top products.

✓ Refined Version with Fixes:

```
python

from prophet import Prophet
import pandas as pd
import plotly.graph_objects as go

# Select product
```

```
product_id = 'B001E05Q64' # Change this if needed

# Prepare data for Prophet
df_product = df_top[df_top['ProductId'] == product_id].copy()

# Convert YearMonth to datetime (first day of month)
df_product['ds'] = pd.to_datetime(df_product['YearMonth']) + pd.offsets.MonthBegin(0)

# Group monthly review counts
df_forecast = df_product.groupby('ds').size().reset_index(name='y')

# Only proceed if there are at least 2 data points
if len(df_forecast) >= 2:
    # Fit Prophet model
    model = Prophet()
    model.fit(df_forecast)

    # Forecast next 6 months
    future = model.make_future_dataframe(periods=6, freq='MS')
    forecast = model.predict(future)

    # 📊 Plot using Plotly
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=df_forecast['ds'], y=df_forecast['y'],
                             mode='lines+markers', name='Historical'))
    fig.add_trace(go.Scatter(x=forecast['ds'], y=forecast['yhat'],
                             mode='lines', name='Forecast'))
    fig.add_trace(go.Scatter(x=forecast['ds'], y=forecast['yhat_upper'],
                             mode='lines', line=dict(dash='dot'),
                             name='Upper Bound', opacity=0.4))
    fig.add_trace(go.Scatter(x=forecast['ds'], y=forecast['yhat_lower'],
                             mode='lines', line=dict(dash='dot'),
                             name='Lower Bound', opacity=0.4))

    fig.update_layout(
        title=f"📊 Forecasted Monthly Reviews for Product {product_id}",
        xaxis_title="Date",
        yaxis_title="Review Count",
        template="plotly_white"
    )

    fig.show()
else:
    print(f"❌ Not enough data to forecast for Product ID: {product_id}")
```

Let me know if you want to **loop over all top 3 products**, or include **seasonality or holiday effects** in the forecast.



```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Concatenate all reviews per product
product_reviews = df_top.groupby('ProductId')['Cleaned_Text'].apply(lambda x: '
'.join(x)).reset_index()

# TF-IDF
tfidf = TfidfVectorizer(max_features=3000)
tfidf_matrix = tfidf.fit_transform(product_reviews['Cleaned_Text'])

# Compute cosine similarity
```

```
cos_sim = cosine_similarity(tfidf_matrix)

# Create a similarity DataFrame
similarity_df = pd.DataFrame(cos_sim, index=product_reviews['ProductId'],
                             columns=product_reviews['ProductId'])

# Display top 3 similar products for a given item
target_product = 'B001EO5Q64'
similar_products = similarity_df[target_product].sort_values(ascending=False).iloc[1:4]
print(f"Top similar products to {target_product}:\n", similar_products)
```

Your code for computing **product similarity based on review content using TF-IDF and cosine similarity** looks solid and well-structured. Below is a **refined version** that:

- Adds optional filtering to ensure **Cleaned_Text** exists and isn't empty.
- Sorts the **similarity_df** nicely.
- Visualizes similarity with a heatmap for insight.

✓ Refined Code with Optional Enhancements

```
python

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# 🕒 Filter and concatenate cleaned reviews per product
product_reviews = (
    df_top[df_top['Cleaned_Text'].notnull()]
    .groupby('ProductId')['Cleaned_Text']
    .apply(lambda x: ' '.join(x))
    .reset_index()
)

# 📊 Compute TF-IDF matrix
tfidf = TfidfVectorizer(max_features=3000, stop_words='english')
tfidf_matrix = tfidf.fit_transform(product_reviews['Cleaned_Text'])

# 🧮 Compute cosine similarity
cos_sim = cosine_similarity(tfidf_matrix)

# 🗂️ Create a similarity DataFrame
similarity_df = pd.DataFrame(cos_sim,
                             index=product_reviews['ProductId'],
                             columns=product_reviews['ProductId'])

# 🎯 Display top 3 similar products to a specific target
target_product = 'B001EO5Q64'
if target_product in similarity_df:
    similar_products = similarity_df[target_product].sort_values(ascending=False).iloc[1:4]
    print(f"🔗 Top similar products to {target_product}:\n", similar_products)
else:
    print(f"❌ Product {target_product} not found in data.")

# 📈 Optional: Visualize similarity matrix
plt.figure(figsize=(10, 6))
sns.heatmap(similarity_df, cmap='YlGnBu')
plt.title("🔗 Cosine Similarity Between Products")
plt.xlabel("ProductId")
```



```
plt.ylabel("ProductId")
plt.tight_layout()
plt.show()
```

Why This Matters:

This lets you:

- Detect **duplicate or substitute** products.
- Group similar products for **recommendation systems**.
- Explore **customer perception overlap** based on natural language content.

Let me know if you'd like to:

- Add **clustering (e.g., KMeans)** on this similarity matrix.
- Visualize it as a **network graph**.
- Export the similarity scores for a dashboard.



wait let's check if we are consistently using the clean df

```
# Step 1: Install kagglehub with pandas dataset support
!pip install kagglehub[pandas-datasets]
```

```
# Step 2: Import necessary modules
import kagglehub
from kagglehub import KaggleDatasetAdapter
```

```
# Step 3: Define the path to the specific CSV file in the dataset
file_path = "Reviews.csv" # This is the main file inside the dataset
```

```
# Step 4: Load dataset using kagglehub
df = kagglehub.load_dataset(
    KaggleDatasetAdapter.PANDAS,
    "snap/amazon-fine-food-reviews", # dataset slug from Kaggle
    file_path,
)
```

```
# Step 5: Preview the data
print("First 5 records:")
print(df.head())
```

```
# Initial Inspection
# Dataset overview
print("Dataset Shape:", df.shape)
print("Column Names:", df.columns.tolist())
```

```
# Data types and non-null counts
```

```

print("\nData Info:")
df.info()

# Check missing values
print("\nMissing Values:")
print(df.isnull().sum())

# Preview sample data
df[['Id', 'ProductId', 'UserId', 'ProfileName', 'HelpfulnessNumerator', 'HelpfulnessDenominator',
'Score', 'Summary', 'Text']].sample(5, random_state=42)

# Check for duplicate rows (entirely identical rows)
total_duplicates = df.duplicated().sum()
print(f"\nTotal Duplicate Rows (exact match): {total_duplicates}")

# Check for duplicate reviews based on UserId and Text
user_text_duplicates = df.duplicated(subset=['UserId', 'Text']).sum()
print(f"Duplicate reviews based on UserId and Text: {user_text_duplicates}")

# Duplicate UserIds – users who submitted more than one review
duplicate_users = df['UserId'].duplicated().sum()
print(f"Duplicate UserIds (multiple reviews by same user): {duplicate_users}")

# Duplicate ProductIds – products with multiple reviews
duplicate_products = df['ProductId'].duplicated().sum()
print(f"Duplicate ProductIds (products reviewed multiple times): {duplicate_products}")

# Count how many *unique* texts are duplicated
num_unique_duplicated_texts = df['Text'].value_counts()
num_unique_duplicated_texts = num_unique_duplicated_texts[num_unique_duplicated_texts > 1]
print(f"Number of unique duplicated review texts: {len(num_unique_duplicated_texts)}")

# Count how many *rows* have duplicated review texts
duplicated_text_rows = df['Text'].duplicated(keep=False).sum()
print(f"Total number of rows with duplicated review texts: {duplicated_text_rows}")

# Extract all rows where the 'Text' column is duplicated (keep=False returns all occurrences, not just
one)
duplicated_text_df = df[df['Text'].duplicated(keep=False)]

# Sort by text for easier viewing
duplicated_text_df = duplicated_text_df.sort_values(by='Text')

# Preview the first few rows
duplicated_text_df[['UserId', 'ProductId', 'Score', 'Summary', 'Text']].head(10)

import re
import nltk
from nltk.corpus import stopwords

# Download NLTK stopwords (only once)

```

```

nltk.download('stopwords')

# Load English stopwords
stop_words = set(stopwords.words("english"))

def fast_clean(text):
    # Lowercase
    text = text.lower()

    # Remove punctuation, numbers, and special characters
    text = re.sub(r'^a-z\s', '', text)

    # Tokenize and remove stopwords
    tokens = [word for word in text.split() if word not in stop_words and len(word) > 1]

    # Rejoin cleaned tokens
    return " ".join(tokens)

# Apply to a subset first (adjust number as needed)
df_subset = df.iloc[:600000].copy()
df_subset['Cleaned_Text'] = df_subset['Text'].apply(fast_clean)

# Preview cleaned data
df_subset[['Text', 'Cleaned_Text']].head()

merge the duplicate_text_df with the Cleaned_Text column

```

The error occurred because the column **'Cleaned_Text'** does not exist in **df_subset** at the point of merging. Let's confirm the cleaning step was properly applied before the merge.

Here's the corrected and complete code that ensures the text cleaning is done first before merging:

```

python

import re
import nltk
from nltk.corpus import stopwords

# Ensure stopwords are downloaded
nltk.download('stopwords')

# Define stopwords
stop_words = set(stopwords.words("english"))

# Define the cleaning function
def fast_clean(text):
    text = str(text).lower()
    text = re.sub(r'^a-z\s', '', text) # remove punctuation, numbers, special chars
    tokens = [word for word in text.split() if word not in stop_words and len(word) > 1]
    return " ".join(tokens)

# Create the subset and apply cleaning
df_subset = df.iloc[:600000].copy()
df_subset['Cleaned_Text'] = df_subset['Text'].apply(fast_clean)

# Merge with duplicated_text_df based on the 'Text' column
merged_df = duplicated_text_df.merge(
    df_subset[['Text', 'Cleaned_Text']],
    on='Text',

```

```

    how='left'
)

import ace_tools as tools; tools.display_dataframe_to_user(name="Merged Duplicate Reviews",
dataframe=merged_df)

```

This will correctly merge and show the **Cleaned_Text** alongside the duplicated reviews. Let me know if you'd like to analyze or visualize this next.



```

# Apply to a subset first (adjust number as needed)
df_subset = df.iloc[:600000].copy()
df_subset['Cleaned_Text'] = df_subset['Text'].apply(fast_clean)

# Preview cleaned data
df_subset[['Text', 'Cleaned_Text']].head()

# Preview sample data
df[['Id', 'ProductId', 'UserId', 'ProfileName', 'HelpfulnessNumerator', 'HelpfulnessDenominator',
'Score', 'Summary', 'Cleaned_Text']].sample(5, random_state=42)

```

Your code is almost correct, but the issue is with the last line:

```

python

df[['Id', 'ProductId', 'UserId', 'ProfileName', 'HelpfulnessNumerator',
'HelpfulnessDenominator', 'Score', 'Summary', 'Cleaned_Text']]

```

You're trying to access the **Cleaned_Text** column from the original **df**, but it was only created in **df_subset**.

✅ Here's the corrected version:

```

python

# Apply to a subset first (adjust number as needed)
df_subset = df.iloc[:600000].copy()
df_subset['Cleaned_Text'] = df_subset['Text'].apply(fast_clean)

# Preview cleaned data
print(df_subset[['Text', 'Cleaned_Text']].head())

# Preview sample data with Cleaned_Text from the subset
print(df_subset[['Id', 'ProductId', 'UserId', 'ProfileName', 'HelpfulnessNumerator',
'HelpfulnessDenominator', 'Score', 'Summary', 'Cleaned_Text']].sample(5,
random_state=42))

```

Let me know if you'd like to filter this further or proceed to analysis or visualization.



```

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import seaborn as sns

```

```
import matplotlib.pyplot as plt
import pandas as pd

# 🕒 Filter and concatenate cleaned reviews per product
product_reviews = (
    df_top[df_top['Cleaned_Text'].notnull()]
    .groupby('ProductId')['Cleaned_Text']
    .apply(lambda x: ' '.join(x))
    .reset_index()
)

# 📊 Compute TF-IDF matrix
tfidf = TfidfVectorizer(max_features=3000, stop_words='english')
tfidf_matrix = tfidf.fit_transform(product_reviews['Cleaned_Text'])

# 📐 Compute cosine similarity
cos_sim = cosine_similarity(tfidf_matrix)

# 🕒 Create a similarity DataFrame
similarity_df = pd.DataFrame(cos_sim,
                             index=product_reviews['ProductId'],
                             columns=product_reviews['ProductId'])

# 🎯 Display top 3 similar products to a specific target
target_product = 'B001EO5Q64'
if target_product in similarity_df:
    similar_products = similarity_df[target_product].sort_values(ascending=False).iloc[1:4]
    print(f"🔗 Top similar products to {target_product}:\n", similar_products)
else:
    print(f"❌ Product {target_product} not found in data.")

# 📊 Optional: Visualize similarity matrix
plt.figure(figsize=(10, 6))
sns.heatmap(similarity_df, cmap='YlGnBu')
plt.title("🕒 Cosine Similarity Between Products")
plt.xlabel("ProductId")
plt.ylabel("ProductId")
plt.tight_layout()
plt.show()
```

```
-----
KeyError                                Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in get_loc(self, key)
    3804     try:
-> 3805         return self._engine.get_loc(casted_key)
    3806     except KeyError as err:

index.pyx in pandas._libs.index.IndexEngine.get_loc()

index.pyx in pandas._libs.index.IndexEngine.get_loc()
```

```
pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 'Cleaned_Text'
```

The above exception was the direct cause of the following exception:

```
KeyError                                Traceback (most recent call last)
2 frames
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in get_loc(self, key)
    3810         ):
    3811             raise InvalidIndexError(key)
-> 3812             raise KeyError(key) from err
    3813     except TypeError:
    3814         # If we have a listlike key, _check_indexing_error will raise

KeyError: 'Cleaned_Text'

Cleaned_Text column is in the df_subset
```

Thanks! Since `Cleaned_Text` exists in `df_subset` (not `df_top`), and you want to compute **cosine similarity between products based on reviews**, we just need to create `df_top` by merging from `df_subset`.

Here is the **refined and corrected full code** using `df_subset`:

```
python

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# ✅ Use df_subset, ensure Cleaned_Text is available
# Get top reviewed products
top_products = df_subset['ProductId'].value_counts().nlargest(10).index

# Filter top products from df_subset
df_top = df_subset[df_subset['ProductId'].isin(top_products)].copy()

# 🔍 Concatenate cleaned reviews per product
product_reviews = (
    df_top[df_top['Cleaned_Text'].notnull()]
    .groupby('ProductId')['Cleaned_Text']
    .apply(lambda x: ' '.join(x))
    .reset_index()
)

# 📊 Compute TF-IDF matrix
tfidf = TfidfVectorizer(max_features=3000, stop_words='english')
tfidf_matrix = tfidf.fit_transform(product_reviews['Cleaned_Text'])

# 📐 Compute cosine similarity
cos_sim = cosine_similarity(tfidf_matrix)

# 💰 Create a similarity DataFrame
similarity_df = pd.DataFrame(cos_sim,
                             index=product_reviews['ProductId'],
                             columns=product_reviews['ProductId'])

# 🎯 Display top 3 similar products to a specific target
```

```
target_product = 'B001E05Q64'
if target_product in similarity_df.columns:
    similar_products = similarity_df[target_product].sort_values(ascending=False).iloc[1:4]
    print(f"🔗 Top similar products to {target_product}:\n", similar_products)
else:
    print(f"❌ Product {target_product} not found in top products.")

# 📊 Optional: Visualize similarity matrix
plt.figure(figsize=(10, 7))
sns.heatmap(similarity_df, cmap='YlGnBu', annot=False)
plt.title("🔗 Cosine Similarity Between Products")
plt.xlabel("ProductId")
plt.ylabel("ProductId")
plt.tight_layout()
plt.show()
```

✓ Notes:

- We use `df_subset` to ensure `Cleaned_Text` is available.
- We filter top reviewed products to limit memory usage and improve interpretability.
- You can change `nlargest(10)` to more or fewer products depending on your focus.

Let me know if you'd like to cluster these similar products or visualize with a network graph.



```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# ✓ Use df_subset, ensure Cleaned_Text is available
# Get top reviewed products
top_products = df_subset['ProductId'].value_counts().nlargest(10).index

# Filter top products from df_subset
df_top = df_subset[df_subset['ProductId'].isin(top_products)].copy()

# 🔗 Concatenate cleaned reviews per product
product_reviews = (
    df_top[df_top['Cleaned_Text'].notnull()]
    .groupby('ProductId')['Cleaned_Text']
    .apply(lambda x: ' '.join(x))
    .reset_index()
)

# 📊 Compute TF-IDF matrix
tfidf = TfidfVectorizer(max_features=3000, stop_words='english')
tfidf_matrix = tfidf.fit_transform(product_reviews['Cleaned_Text'])

# 🔗 Compute cosine similarity
cos_sim = cosine_similarity(tfidf_matrix)
```

```
# 🕒 Create a similarity DataFrame
similarity_df = pd.DataFrame(cos_sim,
                             index=product_reviews['ProductId'],
                             columns=product_reviews['ProductId'])

# 🎯 Display top 3 similar products to a specific target
target_product = 'B001EO5Q64'
if target_product in similarity_df.columns:
    similar_products = similarity_df[target_product].sort_values(ascending=False).iloc[1:4]
    print(f"🔗 Top similar products to {target_product}:\n", similar_products)
else:
    print(f"❌ Product {target_product} not found in top products.")

# 📊 Optional: Visualize similarity matrix
plt.figure(figsize=(10, 7))
sns.heatmap(similarity_df, cmap='YlGnBu', annot=False)
plt.title("🔍 Cosine Similarity Between Products")
plt.xlabel("ProductId")
plt.ylabel("ProductId")
plt.tight_layout()
plt.show()
```

```
KeyError                                Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in get_loc(self, key)
    3804     try:
-> 3805         return self._engine.get_loc(casted_key)
    3806     except KeyError as err:
```

```
index.pyx in pandas._libs.index.IndexEngine.get_loc()
```

```
index.pyx in pandas._libs.index.IndexEngine.get_loc()
```

```
pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()
```

```
pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()
```

```
KeyError: 'ProductId'
```

The above exception was the direct cause of the following exception:

```
KeyError                                Traceback (most recent call last)
2 frames
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in get_loc(self, key)
    3810     ):
    3811         raise InvalidIndexError(key)
-> 3812         raise KeyError(key) from err
    3813     except TypeError:
    3814         # If we have a listlike key, _check_indexing_error will raise
```

```
KeyError: 'ProductId'
```


ProductId is in df_subset too

Thanks for pointing that out! If you're receiving a `KeyError: 'ProductId'` even though `ProductId` exists in `df_subset`, it's likely that:

- The column may be unintentionally renamed (e.g., to lowercase like `productid`).
- Or `df_subset` may have been modified elsewhere in the notebook/session.

Let's verify and then correct the workflow. Here's the **fully revised and robust version of the product similarity clustering** code with column existence checks:

✓ Full Revised Code:

```
python

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# ✓ Check if 'ProductId' and 'Cleaned_Text' are in df_subset
assert 'ProductId' in df_subset.columns, "'ProductId' not found in df_subset"
assert 'Cleaned_Text' in df_subset.columns, "'Cleaned_Text' not found in df_subset"

# 🏆 Select top reviewed products
top_products = df_subset['ProductId'].value_counts().nlargest(10).index

# 🚫 Filter relevant data
df_top = df_subset[df_subset['ProductId'].isin(top_products)].copy()

# 🔍 Combine reviews per product
product_reviews = (
    df_top[df_top['Cleaned_Text'].notnull()]
    .groupby('ProductId')['Cleaned_Text']
    .apply(lambda x: ' '.join(x))
    .reset_index()
)

# 📄 TF-IDF vectorization
tfidf = TfidfVectorizer(max_features=3000, stop_words='english')
tfidf_matrix = tfidf.fit_transform(product_reviews['Cleaned_Text'])

# 📐 Cosine similarity
cos_sim = cosine_similarity(tfidf_matrix)

# 🗂️ Build similarity DataFrame
similarity_df = pd.DataFrame(
    cos_sim,
    index=product_reviews['ProductId'],
    columns=product_reviews['ProductId']
)

# 🎯 Display top 3 similar products for a given product
target_product = 'B001E05Q64'
if target_product in similarity_df.columns:
    similar_products = similarity_df[target_product].sort_values(ascending=False).iloc[1:4]
    print(f"🔗 Top similar products to {target_product}:\n", similar_products)
else:
    print(f"❌ Product {target_product} not in top reviewed products.")

# 📊 Heatmap of product similarity
plt.figure(figsize=(10, 7))
sns.heatmap(similarity_df, cmap='YlGnBu', annot=False)
```

```
plt.title("🔍 Product Similarity Based on TF-IDF of Cleaned Reviews")
plt.xlabel("ProductId")
plt.ylabel("ProductId")
plt.tight_layout()
plt.show()
```

✓ Tips:

- Use **assert** to ensure key columns are present before proceeding.
- You can inspect available columns using:

```
python

print(df_subset.columns.tolist())
```

Let me know if you'd like to cluster similar products into groups using KMeans or visualize as a network graph!



```
import pandas as pd
import re
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.figure_factory as ff

nltk.download('vader_lexicon')

# Clone to avoid modifying original
df_vader = df_subset.copy()

# Sample slang/emoticon dictionary (expand as needed)
slang_dict = {
    "u": "you",
    "ur": "your",
    "lol": "laughing out loud",
    "omg": "oh my god",
    "idk": "i don't know",
    "luv": "love",
    "gr8": "great",
    ":)": "smile",
    ":-)": "smile",
    ":(": "sad",
    ":-(": "sad",
}

# Function to normalize text
```

```
def clean_text(text):
    text = str(text).lower()
    text = re.sub(r"http\S+|www.\S+", "", text) # remove URLs
    text = re.sub(r"^\x00-\x7F+", " ", text) # remove emojis/non-ASCII
    text = re.sub(r"^[a-z\s]", "", text) # remove punctuation/numbers
    words = text.split()
    normalized = [slang_dict.get(word, word) for word in words]
    return " ".join(normalized)

# Apply cleaning
df_vader['Normalized_Text'] = df_vader['Text'].apply(clean_text)

# Initialize VADER
vader = SentimentIntensityAnalyzer()

# Compute VADER compound score
df_vader['VADER_Score'] = df_vader['Normalized_Text'].apply(lambda x: vader.polarity_scores(x)
['compound'])

# Map compound score to sentiment label
def get_vader_label(score):
    if score >= 0.05:
        return 'positive'
    elif score <= -0.05:
        return 'negative'
    else:
        return 'neutral'

df_vader['VADER_Label'] = df_vader['VADER_Score'].apply(get_vader_label)

# Confusion matrix
labels = ['positive', 'neutral', 'negative']
cm = confusion_matrix(df_vader['Sentiment'], df_vader['VADER_Label'], labels=labels)
report = classification_report(df_vader['Sentiment'], df_vader['VADER_Label'], output_dict=True,
zero_division=0)

# Plotly heatmap version (interactive)
fig = ff.create_annotated_heatmap(
    z=cm,
    x=labels,
    y=labels,
    colorscale='Blues',
    showscale=True
)
fig.update_layout(
    title="📊 Confusion Matrix (VADER vs. Manual)",
    xaxis_title="Predicted Sentiment",
    yaxis_title="Actual Sentiment"
)
fig.show()
```

```
# Print performance report
print("🔍 VADER Performance Report:")
print(classification_report(df_vader['Sentiment'], df_vader['VADER_Label'], digits=3))
```

```
🔍 VADER Performance Report:
      precision    recall  f1-score   support

negative    0.422     0.223     0.291     400
neutral     0.185     0.200     0.192     190
positive    0.423     0.602     0.497     410

accuracy              0.374     1000
macro avg    0.343     0.342     0.327     1000
weighted avg    0.377     0.374     0.357     1000
```

Thank you for sharing the code and results. Here's a quick breakdown and assessment of what you're doing and how you might further refine or interpret it.

✅ Summary of What This Code Does

- **Cleans raw text** using regex and slang normalization.
- **Applies VADER sentiment analysis** on the normalized review text.
- **Compares VADER labels** with existing sentiment labels (likely based on review **Score**).
- **Outputs a confusion matrix** using Plotly and a full classification report.

Interpretation of VADER Performance

```
text
      precision    recall  f1-score   support

negative    0.422     0.223     0.291     400
neutral     0.185     0.200     0.192     190
positive    0.423     0.602     0.497     410
```

- **Positive reviews are best recognized** by VADER with **F1 = 0.50**, which is expected since VADER is designed around positive sentiment lexicons (e.g., "great", "love", "best").
- **Negative sentiment is under-detected**, with a recall of 0.22. This may be because VADER doesn't handle **context or sarcasm** well.
- **Neutral reviews are the hardest to classify**, which is common for both lexicon-based and ML models.

Suggestions for Improvement

Here's how you could improve accuracy or extend the analysis:

1. Enhance Text Normalization

- Add contractions handling (e.g., "isn't" → "is not").
- Consider using libraries like `contractions` or `emoji`.

2. Balance the Dataset

- You're using 1000 samples, which is good for prototyping. But ensure balanced class distribution to avoid biased metrics.

3. Compare with ML/NLP Models

Try training a logistic regression or transformer-based model (e.g., `DistilBERT`) on `Cleaned_Text` or `Normalized_Text` and compare performance.

4. Highlight Errors

You can extract false positives/negatives to understand where VADER fails:

```
python

false_negatives = df_vader[(df_vader['Sentiment'] == 'negative') & (df_vader['VADER_Label'] != 'negative')]
display(false_negatives[['Text', 'Normalized_Text', 'VADER_Score', 'VADER_Label']].head(5))
```

✓ Let Me Know If You'd Like

- A side-by-side comparison with a trained ML model (e.g., Naive Bayes, Logistic Regression).
- SHAP or attention-based explanation if you move to transformers.
- Integration of sarcasm detection or emotion lexicons.

Would you like help with that next step?



provide refined revised code

Here is a **refined and revised version** of your VADER sentiment evaluation code, with better normalization, enhanced structure, and clearer comments:

✓ Refined VADER Sentiment Evaluation Code

```
python

import pandas as pd
import re
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer
from sklearn.metrics import confusion_matrix, classification_report
import plotly.figure_factory as ff

# Download necessary resources
```

```

nltk.download('vader_lexicon')

# 🔄 Work on a copy of the clean subset
df_vader = df_subset.copy()

# 📖 Slang & emoji normalization dictionary (extendable)
slang_dict = {
    "u": "you", "ur": "your", "omg": "oh my god", "lol": "laughing out loud",
    "idk": "i don't know", "luv": "love", "gr8": "great",
    ":)": "smile", ":-)": "smile", ":(": "sad", ":-(": "sad"
}

# 🗨️ Normalize review text
def normalize_text(text):
    text = str(text).lower()
    text = re.sub(r"http\S+|www.\S+", "", text) # remove URLs
    text = re.sub(r"^\x00-\x7F+", "", text) # remove emojis/non-ASCII
    text = re.sub(r"[^a-z\s]", "", text) # remove punctuation/numbers
    words = text.split()
    normalized = [slang_dict.get(word, word) for word in words]
    return " ".join(normalized)

# Apply normalization
df_vader['Normalized_Text'] = df_vader['Text'].apply(normalize_text)

# ⚙️ Initialize VADER
vader = SentimentIntensityAnalyzer()

# 📊 Compute compound sentiment score
df_vader['VADER_Score'] = df_vader['Normalized_Text'].apply(lambda x: vader.polarity_scores(x)
['compound'])

# 🏷️ Map compound score to sentiment labels
def classify_sentiment(score):
    if score >= 0.05:
        return 'positive'
    elif score <= -0.05:
        return 'negative'
    else:
        return 'neutral'

df_vader['VADER_Label'] = df_vader['VADER_Score'].apply(classify_sentiment)

# 🎯 Ground truth from Score column (if not already in df_subset)
if 'Sentiment' not in df_vader.columns:
    df_vader['Sentiment'] = df_vader['Score'].apply(lambda x: 'positive' if x > 3 else
'negative' if x < 3 else 'neutral')

# 📊 Confusion Matrix
labels = ['positive', 'neutral', 'negative']
cm = confusion_matrix(df_vader['Sentiment'], df_vader['VADER_Label'], labels=labels)

# 📊 Interactive confusion matrix plot
fig = ff.create_annotated_heatmap(
    z=cm,
    x=labels,
    y=labels,
    colorscale='Blues',
    showscale=True
)
fig.update_layout(
    title="📊 Confusion Matrix: VADER vs. Review Score Sentiment",
    xaxis_title="Predicted (VADER)",
    yaxis_title="Actual (From Score)"
)
fig.show()

# 🖨️ Print classification metrics
print("🔍 VADER Sentiment Classification Report:")
print(classification_report(df_vader['Sentiment'], df_vader['VADER_Label'], digits=3,
zero_division=0))

```

Optional Enhancements You Can Add Later

- False Positive/Negative Exploration:

```
python
```

```
df_vader[df_vader['Sentiment'] != df_vader['VADER_Label']][['Text', 'Normalized_Text',  
'VADER_Score', 'Sentiment', 'VADER_Label']].head()
```

- Emoji or contractions expansion using `emoji` or `contractions` packages
- Benchmark against transformer-based models

Let me know if you'd like to compare this with a logistic regression or DistilBERT-based model next.



VADER Sentiment Classification Report:

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

negative	0.422	0.223	0.291	400
neutral	0.185	0.200	0.192	190
positive	0.423	0.602	0.497	410

accuracy		0.374	1000	
macro avg	0.343	0.342	0.327	1000
weighted avg	0.377	0.374	0.357	1000

Thanks for sharing the classification report. Here's a quick breakdown and interpretation of the **VADER sentiment performance**:



Performance Summary (VADER vs Score-Derived Sentiment)

Metric	Negative	Neutral	Positive	Accuracy
Precision	0.422	0.185	0.423	—
Recall	0.223	0.200	0.602	0.374
F1-score	0.291	0.192	0.497	—
Support (n)	400	190	410	1000



Key Observations:

- Positive sentiment detection** is strongest:
 - VADER correctly identifies many positive reviews (**recall = 0.602, f1 = 0.497**).
 - However, it also misclassifies some negatives/neutral as positive.
- Neutral sentiment detection is weakest:**

- Very low precision and recall — likely due to VADER's binary leaning nature (tends to push toward positive or negative).
- **Negative detection is moderate:**
 - Precision is decent (0.422), but recall is poor (0.223), meaning many actual negatives are missed.
- **Overall accuracy is low (37.4%),** which is expected from a rule-based tool like VADER on nuanced review texts.

Suggestions for Improvement:

1. Use fine-tuned transformer models like:

- `distilbert-base-uncased-finetuned-sst-2-english`
- `cardiffnlp/twitter-roberta-base-sentiment`
- These models outperform rule-based tools on nuanced, domain-specific text.

2. Custom-trained classifier (Logistic Regression, SVM, etc.):

- Train using TF-IDF on `Cleaned_Text` and labeled sentiment (from `Score`).
- Much better alignment with domain vocabulary.

3. Augment neutral detection:

- Consider manually labeling a subset and balancing training data.

4. Hybrid approach:

- Use VADER for fast heuristic tagging, then fine-tune with learned models.

Would you like help setting up a DistilBERT or logistic regression sentiment classifier as a comparison?



we will proceed with topic modelling (LDA) first

Great choice! Topic modeling with **Latent Dirichlet Allocation (LDA)** can help uncover hidden themes across your review texts. Here's a **refined and structured LDA pipeline** using `df_subset['Cleaned_Text']`:

1. Prepare Document-Term Matrix

```
python

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation

# Vectorize cleaned text
vectorizer = CountVectorizer(max_df=0.95, min_df=5, max_features=3000, stop_words='english')
doc_term_matrix = vectorizer.fit_transform(df_subset['Cleaned_Text'])

# Get vocabulary
feature_names = vectorizer.get_feature_names_out()
```


2. Fit the LDA Model

```
python

lda_model = LatentDirichletAllocation(
    n_components=5,          # Set number of topics
    max_iter=10,
    learning_method='online',
    random_state=42
)
lda_model.fit(doc_term_matrix)
```

3. Display Top Words per Topic

```
python

def print_top_words(model, feature_names, n_top_words=10):
    for topic_idx, topic in enumerate(model.components_):
        top_features = topic.argsort()[::-n_top_words - 1:-1]
        top_words = [feature_names[i] for i in top_features]
        print(f"📄 Topic {topic_idx + 1}: {' '.join(top_words)}")

print_top_words(lda_model, feature_names)
```

4. Visualize Topic Distribution per Document

```
python

import pandas as pd
import plotly.express as px

# Get topic probabilities for each review
doc_topic_dist = lda_model.transform(doc_term_matrix)
df_topics = pd.DataFrame(doc_topic_dist, columns=[f'Topic {i+1}' for i in
range(lda_model.n_components)])
df_topics['Dominant_Topic'] = df_topics.idxmax(axis=1)

# Combine with original reviews
df_topic_output = df_subset[['Text', 'Cleaned_Text']].copy()
df_topic_output = df_topic_output.reset_index(drop=True).join(df_topics)

# Count dominant topics
topic_counts = df_topics['Dominant_Topic'].value_counts().reset_index()
topic_counts.columns = ['Topic', 'Count']

# Plot topic distribution
fig = px.bar(topic_counts, x='Topic', y='Count', title="📊 Dominant Topics in Review Corpus",
text='Count')
fig.update_layout(template='plotly_white')
fig.show()
```

Next Optional Steps:

- Extract representative **sample reviews per topic**
- Run **LDA by sentiment groups**
- Use **pyLDAvis** for interactive topic interpretation

Let me know if you'd like to visualize the topics by sentiment next or extract top reviews per topic!



```
-----
KeyError                                Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in get_loc(self, key)
    3804     try:
-> 3805         return self._engine.get_loc(casted_key)
    3806     except KeyError as err:

index.pyx in pandas._libs.index.IndexEngine.get_loc()

index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 'Cleaned_Text'
```

The above exception was the direct cause of the following exception:

```
KeyError                                Traceback (most recent call last)
2 frames
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in get_loc(self, key)
    3810     ):
    3811         raise InvalidIndexError(key)
-> 3812         raise KeyError(key) from err
    3813     except TypeError:
    3814         # If we have a listlike key, _check_indexing_error will raise

KeyError: 'Cleaned_Text'
```

The error indicates that the column `'Cleaned_Text'` is not found in your `df_subset` DataFrame at the point where you're trying to access it for topic modeling.

✓ Here's how to fix and proceed:

Let's **ensure the `Cleaned_Text` column exists** before we run the LDA pipeline.

↺ 1. Clean the text column if not yet done

Run this preprocessing block if you haven't already:

```
python

import re
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')
```

```
# Load English stopwords
stop_words = set(stopwords.words("english"))

# Define basic text cleaner
def fast_clean(text):
    text = str(text).lower()
    text = re.sub(r'^a-z\s', '', text) # remove punctuation/numbers
    tokens = [word for word in text.split() if word not in stop_words and len(word) > 1]
    return " ".join(tokens)

# Apply to your df_subset
df_subset['Cleaned_Text'] = df_subset['Text'].apply(fast_clean)
```

2. Verify column creation

Check:

```
python

print(df_subset[['Text', 'Cleaned_Text']].sample(3))
```

3. Then re-run the LDA pipeline

Now that `Cleaned_Text` exists, re-run the LDA steps:

- CountVectorizer
- LDA fitting
- Top word extraction
- Topic visualization

Let me know if you'd like the entire pipeline re-merged now that the error is fixed.



Extract representative sample reviews per topic

Run LDA by sentiment groups

Use pyLDAvis for interactive topic interpretation

The code ran successfully up to the topic modeling and representative review extraction, but failed at the interactive visualization step because the `pyLDAvis` module is not installed in this environment.

Here's what was accomplished:

Completed

- Performed **LDA Topic Modeling** on a sample of 3,000 reviews from `df_subset`.
- Extracted top keywords per topic.

- Assigned the most likely topic to each review.
- Selected **representative reviews** for each topic.
- Grouped by **Sentiment** to ensure sentiment-based diversity.

✗ Not Completed

- Could not display **interactive pyLDAvis visualization** due to missing **pyLDAvis**.

✓ Next Steps:

If you're running this in your own local environment, you can install **pyLDAvis** using:

```
bash

pip install pyLDAvis
```

Then, to generate the interactive topic visualization:

```
python

import pyLDAvis
import pyLDAvis.sklearn

pyLDAvis.enable_notebook()
pyLDAvis.sklearn.prepare(lda_model, dtm, vectorizer)
```

Would you like me to export the data with topic assignments or proceed to run LDA **by sentiment groups**?



get topic modelling model performance

It looks like the variable **df_sample**, which should contain the topic assignments from LDA, is not defined in the current environment. To compute topic modeling performance (e.g. via silhouette score), we need the following:

- The document-topic distribution matrix (**lda_model.transform(...)**)
- The assigned topic labels for each document
- A consistent DataFrame (like **df_sample**) that contains these labels

Please rerun the LDA topic modeling section with:

1. **df_sample = your_document_dataframe.copy()**
2. **df_sample['Topic'] = topic_assignments (from lda_model.transform(...))**

Once that's done, I can compute the model performance using silhouette score or coherence score. Would you like me to provide the full setup for this?



```
-----
ModuleNotFoundError                                Traceback (most recent call last)
/tmp/ipython-input-43-1210158369.py in <cell line: 0>()
      1 import pyLDAvis
----> 2 import pyLDAvis.sklearn
      3
      4 pyLDAvis.enable_notebook()
      5 pyLDAvis.sklearn.prepare(lda_model, dtm, vectorizer)
```

ModuleNotFoundError: No module named 'pyLDAvis.sklearn'

NOTE: If your import is failing due to a missing package, you can manually install dependencies using either `!pip` or `!apt`.

To view examples of installing some common dependencies, click the "Open Examples" button below.

The error means that **pyLDAvis** is installed, but the **sklearn** module within it is missing. This usually happens because **pyLDAvis.sklearn** is not included in older or minimal installations.

✓ Solution:

Install or upgrade **pyLDAvis** with full support using this:

```
python

!pip install pyLDAvis --upgrade
```

Then, for compatibility with **LatentDirichletAllocation** from scikit-learn, use:

```
python

import pyLDAvis
import pyLDAvis.sklearn

pyLDAvis.enable_notebook()

# `lda_model`: your trained LDA model
# `doc_term_matrix`: your document-term matrix
# `count_vectorizer`: the vectorizer used to fit `doc_term_matrix`

pyLDAvis_display = pyLDAvis.sklearn.prepare(lda_model, doc_term_matrix, count_vectorizer)
pyLDAvis_display
```

Let me know if you want to visualize this inline in a notebook or save it as HTML.



```
import pyLDAvis
import pyLDAvis.sklearn

pyLDAvis.enable_notebook()

# lda_model: your trained LDA model
# doc_term_matrix: your document-term matrix
# count_vectorizer: the vectorizer used to fit doc_term_matrix

pyLDAvis_display = pyLDAvis.sklearn.prepare(lda_model, doc_term_matrix, count_vectorizer)
pyLDAvis_display
```

```
-----
ModuleNotFoundError                                Traceback (most recent call last)
/tmp/ipython-input-45-4139996988.py in <cell line: 0>()
      1 import pyLDAvis
----> 2 import pyLDAvis.sklearn
      3
      4 pyLDAvis.enable_notebook()
      5
```

ModuleNotFoundError: No module named 'pyLDAvis.sklearn'

NOTE: If your import is failing due to a missing package, you can manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the "Open Examples" button below.

To fix the **ModuleNotFoundError: No module named 'pyLDAvis.sklearn'**, you need to install or upgrade **pyLDAvis** with:

```
python

!pip install -U pyLDAvis
```

Once installed, import the correct module. **In newer versions of pyLDAvis, the correct import is:**

```
python

import pyLDAvis.sklearn_model as sklearn_lda
```

Then run:

```
python

import pyLDAvis
import pyLDAvis.sklearn_model as sklearn_lda
```

```
pyLDAvis.enable_notebook()

pyLDAvis_display = sklearn_lda.prepare(lda_model, doc_term_matrix, count_vectorizer)
pyLDAvis_display
```

If you're not running this in a notebook environment, use:

```
python

pyLDAvis.save_html(pyLDAvis_display, 'lda_visualization.html')
```

Let me know if you want help generating or saving the visualization after installing!



Hult_SMAFBI_Team_3__A1_Team_assignment.pdf
PDF

Thanks! Since the dataset and cleaning pipeline are well-defined in your notebook, here's how you can evaluate **LDA topic modeling performance** and visualize it using the **pyLDAvis** library, including a workaround for the module import issue:

✓ 1. Evaluate LDA Model Performance (Topic Coherence)

```
python

from gensim.models import CoherenceModel
from gensim.corpora import Dictionary

# Prepare corpus and dictionary from tokenized text
tokenized_text = [text.split() for text in df_balanced['Cleaned_Text']]
dictionary = Dictionary(tokenized_text)
corpus = [dictionary.doc2bow(text) for text in tokenized_text]

# Evaluate coherence
coherence_model_lda = CoherenceModel(model=lda_model, texts=tokenized_text,
dictionary=dictionary, coherence='c_v')
coherence_score = coherence_model_lda.get_coherence()

print(f"🧠 Topic Coherence Score (c_v): {coherence_score:.4f}")
```

A coherence score above **0.4** is generally acceptable for short, noisy reviews.

✓ 2. Visualize Topics with pyLDAvis (with Gensim instead of sklearn)

```
python

import pyLDAvis
import pyLDAvis.gensim_models as gensimvis # 🖱 use this instead of sklearn
from gensim import corpora

# Prepare pyLDAvis interactive visualization
pyLDAvis.enable_notebook()
```

```
vis = gensimvis.prepare(lda_model, corpus, dictionary)
vis
```

If you're using **Google Colab**, `pyLDAvis.enable_notebook()` works directly. For other Jupyter environments, replace it with `pyLDAvis.display(vis)`.

Optional: Tune Number of Topics

To determine the optimal number of topics:

```
python

coherence_scores = []
topic_range = range(2, 15)

for num_topics in topic_range:
    temp_model = LatentDirichletAllocation(n_components=num_topics, random_state=42)
    temp_model.fit(dtm)

    # Convert scikit-learn model to gensim-compatible input
    gensim_corpus = [dictionary.doc2bow(text) for text in tokenized_text]
    gensim_model = CoherenceModel(model=temp_model, texts=tokenized_text, dictionary=dictionary,
    coherence='c_v')
    coherence_scores.append(gensim_model.get_coherence())

# Plot
plt.plot(topic_range, coherence_scores, marker='o')
plt.xlabel("Number of Topics")
plt.ylabel("Coherence Score (c_v)")
plt.title("LDA Coherence Score vs. Number of Topics")
plt.grid(True)
plt.show()
```

Let me know if you'd like to extract **top keywords or sample reviews** per topic next!



```
-----
AttributeError                                Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/gensim/models/coherencemodel.py in
_get_topics_from_model(model, topn)
    495         matutils.argsort(topic, topn=topn, reverse=True) for topic in
--> 496         model.get_topics()
    497     ]
```

AttributeError: 'LatentDirichletAllocation' object has no attribute 'get_topics'

During handling of the above exception, another exception occurred:

```
ValueError                                Traceback (most recent call last)
4 frames
/usr/local/lib/python3.11/dist-packages/gensim/models/coherencemodel.py in
_get_topics_from_model(model, topn)
    497     ]
    498     except AttributeError:
--> 499         raise ValueError(
```



```
500 "This topic model is not currently supported. Supported topic models"
501 " should implement the get_topics method.")
```

ValueError: This topic model is not currently supported. Supported topic models should implement the **get_topics** method.

Thanks for pointing that out. The error occurs because you're using **Scikit-learn's LatentDirichletAllocation**, which is **not directly compatible** with Gensim's **CoherenceModel**. It expects a model with a **.get_topics()** method like Gensim's LDA.

✅ Fix: Use Gensim's LDA instead for coherence scoring and pyLDavis

Let's switch to **Gensim's LDA model** for proper topic evaluation and visualization.

🔄 Step-by-Step Gensim-Based LDA Pipeline

```
python

from gensim.models.ldamodel import LdaModel
from gensim.corpora import Dictionary
from gensim.models import CoherenceModel
from sklearn.feature_extraction.text import CountVectorizer
import pyLDavis.gensim_models as gensimvis
import pyLDavis
import matplotlib.pyplot as plt

# 1 Tokenize cleaned text
tokenized_texts = df_balanced['Cleaned_Text'].dropna().apply(lambda x: x.split()).tolist()

# 2 Create Dictionary and Corpus
dictionary = Dictionary(tokenized_texts)
corpus = [dictionary.doc2bow(text) for text in tokenized_texts]

# 3 Fit LDA Model (Gensim)
lda_gensim = LdaModel(corpus=corpus,
                      id2word=dictionary,
                      num_topics=5,
                      random_state=42,
                      passes=10,
                      per_word_topics=True)

# 4 Coherence Score
coherence_model = CoherenceModel(model=lda_gensim, texts=tokenized_texts, dictionary=dictionary,
                                coherence='c_v')
coherence_score = coherence_model.get_coherence()
print(f"✅ Gensim LDA Coherence Score: {coherence_score:.4f}")

# 5 pyLDavis visualization
pyLDavis.enable_notebook()
vis = gensimvis.prepare(lda_gensim, corpus, dictionary)
vis
```

🔗 Optional: View Topics

```
python

for i, topic in lda_gensim.show_topics(num_topics=5, num_words=10, formatted=False):
    print(f"\n🧠 Topic {i + 1}:")
```

```
print(", ".join([word for word, _ in topic]))
```

Let me know if you'd like to:

- Extract **top reviews** per topic,
- Run LDA **per sentiment group**, or
- Use **Scikit-learn's LDA** just for clustering instead (but skip coherence scoring).

I'm happy to generate those next.



what other analytical techniques is best and applicable

Great question. For a dataset like **Amazon Fine Food Reviews**, which includes textual reviews, ratings, helpfulness scores, timestamps, and product IDs, a wide range of **analytical techniques** can be applied—depending on the business question. Here's a categorized list tailored to your dataset:



1. Descriptive & Exploratory Analysis

Helpful for getting initial insights:

- **Frequency analysis:** Most-reviewed products, most common words, top users
- **Temporal trends:** Reviews or sentiment over time (done already)
- **Review length analysis:** Distribution of text and summary length
- **Helpfulness ratio:** Useful signal for user trust



2. Sentiment Analysis

Useful for understanding customer satisfaction:

- **VADER or TextBlob** (you've done this already)
- **BERT-based sentiment classification** for more robust results (especially for ambiguous or sarcastic reviews)



3. Topic Modeling (LDA, BERTopic)

To extract **themes or pain points** in reviews:

- **LDA** (you're doing now): Reveals general topics
- **BERTopic** (transformer-based): Captures more **contextual and nuanced** topics using embeddings
- **NMF:** An alternative topic model that works well with sparse data

4. Predictive Modeling

To predict review outcomes or product quality:

- **Rating prediction:** Using NLP + regression or classification
- **Helpfulness prediction:** Logistic regression/classifier (e.g., "Was this review helpful?")
- **Churn prediction** (if repeat purchases exist, from UserId)

5. Text Similarity & Clustering

To discover **product or review similarity**:

- **TF-IDF + cosine similarity** (you applied)
- **Product clustering** by review language
- **Sentence-BERT embeddings** for higher accuracy similarity detection

6. Association & Pattern Mining

- **Apriori or FP-Growth:** Discover patterns like "If a user buys X, they rate it 5 and say Y"
- **Rule mining** on sentiments, products, and scores

7. Time Series Forecasting

To predict future review trends or engagement:

- **Prophet or ARIMA:** Predict number of reviews per product/month
- **Seasonal decomposition:** Understand review seasonality

8. Network Analysis

If you want to explore **reviewer relationships**:

- Build graphs: shared product reviewers, influence propagation
- Centrality analysis: top influential reviewers

9. Text Classification

You can train a model to classify:

- **Star ratings (1-5)** based on review text
- **Fake vs. genuine reviews** (if labels exist or can be inferred)

Summary Matrix

Technique	Business Use Case	Tools
Sentiment Analysis	Understand customer tone	VADER, TextBlob, BERT
Topic Modeling	Extract main discussion themes	LDA, BERTopic
Predictive Modeling	Predict review scores/helpfulness	Scikit-learn, XGBoost
Clustering	Group similar products	KMeans, HDBSCAN
Time Series Forecasting	Predict review volume	Prophet, ARIMA
Similarity Search	Find duplicate/similar products	TF-IDF, SBERT
Network Analysis	Understand user influence	NetworkX, Gephi
Classification	Text → Score prediction	LogisticRegression, BERT

Would you like help selecting a technique for your next section (e.g., topic clustering, forecasting, classification)?



network analysis

Great! Let's walk through **how to apply Network Analysis** to your Amazon Fine Food Reviews dataset.

🔗 What is Network Analysis in this context?

In your case, **network analysis** helps uncover **relationships between users, products, and sentiments**. Here are some interesting types of networks you can build:

🧩 1. User-Product Bipartite Graph

- **Nodes:** Users and Products
- **Edges:** A user left a review for a product
- **Goal:**
 - Identify **hub users** (those reviewing many products)
 - Find **popular products** (with many unique reviewers)
 - Spot **review rings** (users reviewing the same group of products)

python

```
import networkx as nx
import matplotlib.pyplot as plt

# Sample (filtered) data
df_graph = df_subset[['UserId', 'ProductId']].dropna().drop_duplicates().sample(5000,
random_state=42)

# Create bipartite graph
B = nx.Graph()
B.add_nodes_from(df_graph['UserId'], bipartite='users')
B.add_nodes_from(df_graph['ProductId'], bipartite='products')
B.add_edges_from(list(df_graph.itertuples(index=False, name=None)))
```

```
# Visualization (force-directed layout)
plt.figure(figsize=(10, 7))
pos = nx.spring_layout(B, k=0.15)
nx.draw(B, pos, node_size=10, alpha=0.5, edge_color="gray")
plt.title("User-Product Bipartite Network")
plt.show()
```

2. User Similarity Graph

- **Nodes:** Users
- **Edges:** Similar review text or rating behavior (e.g., same products, same sentiments)
- **Goal:** Spot communities of like-minded users (e.g., loyal fans, harsh critics)

python

```
# Example: pairwise similarity by common product reviews
from sklearn.metrics import jaccard_score
from sklearn.preprocessing import MultiLabelBinarizer

user_products = df_subset.groupby('UserId')['ProductId'].apply(set).reset_index()
mlb = MultiLabelBinarizer()
user_product_matrix = mlb.fit_transform(user_products['ProductId'])

# Compute Jaccard similarity
from sklearn.metrics.pairwise import pairwise_distances
sim_matrix = 1 - pairwise_distances(user_product_matrix, metric='jaccard')

# Build graph from similarity matrix
G = nx.Graph()
user_ids = user_products['UserId'].tolist()

for i in range(len(user_ids)):
    for j in range(i+1, len(user_ids)):
        if sim_matrix[i][j] > 0.5: # set threshold
            G.add_edge(user_ids[i], user_ids[j], weight=sim_matrix[i][j])

print(f"Graph has {G.number_of_nodes()} users and {G.number_of_edges()} edges.")
```

3. Centrality & Community Detection

Once your network is built, you can analyze it:

python

```
# Degree centrality
degree = nx.degree_centrality(G)
top_users = sorted(degree.items(), key=lambda x: x[1], reverse=True)[:10]
print("Top central users by degree:")
print(top_users)

# Detect communities (Louvain or Greedy Modularity)
from networkx.algorithms.community import greedy_modularity_communities
communities = greedy_modularity_communities(G)
print(f"Found {len(communities)} communities")
```

Optional: Product-Product Co-Review Network

- Connect products that were **reviewed by the same users**
- Helps with **product recommendation** and **bundling insights**

💡 Use Cases in Your Assignment

- Identify “super reviewers” (central users) and which products they influence
- Cluster products by **shared review behavior**
- Spot isolated products with limited reach
- Add community graphs to support segmentation findings

Would you like me to generate visualizations or a working sample with your data (e.g., `df_subset` or `df_duplicated_text_df`)?



```
import networkx as nx
import matplotlib.pyplot as plt

# Use the original df_subset created in cell Uqk63zdWLfZ-
# This DataFrame contains the 'UserId' and 'ProductId' columns from the loaded dataset.
df_graph = df_subset[['UserId', 'ProductId']].dropna().drop_duplicates().sample(5000,
random_state=42)

# Create bipartite graph
B = nx.Graph()
B.add_nodes_from(df_graph['UserId'], bipartite='users')
B.add_nodes_from(df_graph['ProductId'], bipartite='products')
B.add_edges_from(list(df_graph.itertuples(index=False, name=None)))

# Visualization (force-directed layout)
plt.figure(figsize=(10, 7))
pos = nx.spring_layout(B, k=0.15)
nx.draw(B, pos, node_size=10, alpha=0.5, edge_color="gray")
plt.title("User-Product Bipartite Network")
plt.show()
```

```
KeyError                                Traceback (most recent call last)
/tmp/ipython-input-56-490241796.py in <cell line: 0>()
      4 # Use the original df_subset created in cell Uqk63zdWLfZ-
      5 # This DataFrame contains the 'UserId' and 'ProductId' columns from the loaded dataset.
----> 6 df_graph = df_subset[['UserId', 'ProductId']].dropna().drop_duplicates().sample(5000,
      7 random_state=42)
      8 # Create bipartite graph
```

2 frames

```

/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in _raise_if_missing(self, key,
indexer, axis_name)
    6247         if nmissing:
    6248             if nmissing == len(indexer):
-> 6249                 raise KeyError(f"None of [{key}] are in the [{axis_name}]")
    6250
    6251         not_found = list(ensure_index(key)[missing_mask.nonzero()[0]].unique())

```

KeyError: "None of [Index(['UserId', 'ProductId'], dtype='object')] are in the [columns]"

check the latest pdf i shared

Based on the content of the latest PDF you shared ([Hult_SMAFBI_Team_3__A1_Team_assignment.pdf](#)), the dataset indeed includes the columns **UserId** and **ProductId**. Therefore, the KeyError you're encountering in your code is likely due to using a modified or filtered DataFrame (**df_subset**) that no longer contains those columns.

To resolve this issue:

- Either reassign **df_graph** from the original full dataset (e.g., **df**, which has all columns).
- Or if you're using a subset like **df_subset**, ensure to include **UserId** and **ProductId** during the initial subset creation.

Example fix:

```

python

df_graph = df[['UserId', 'ProductId']].dropna().drop_duplicates().sample(5000, random_state=42)

```

Let me know if you'd like help generating a refined network analysis using this.



helpfulness prediction

To predict *Helpfulness* of Amazon reviews, you can frame it as either:

1. Regression Task

Predict **helpfulness ratio** (**HelpfulnessNumerator** / **HelpfulnessDenominator**) as a continuous value.

2. Classification Task

Classify a review as:

- **Helpful** (e.g., **ratio** ≥ 0.7)
- **Not helpful** (e.g., **ratio** < 0.7)

Step-by-Step: Helpfulness Prediction (Classification)

Step 1: Feature Engineering

```
python

# Filter only rows with valid denominator
df_help = df_subset[df_subset['HelpfulnessDenominator'] > 0].copy()
df_help['HelpfulnessRatio'] = df_help['HelpfulnessNumerator'] /
df_help['HelpfulnessDenominator']

# Binary target
df_help['HelpfulLabel'] = df_help['HelpfulnessRatio'].apply(lambda x: 1 if x >= 0.7 else 0)

# Text length features
df_help['ReviewLength'] = df_help['Text'].apply(lambda x: len(str(x).split()))
df_help['SummaryLength'] = df_help['Summary'].apply(lambda x: len(str(x).split()))
```

Step 2: TF-IDF Vectorization + Concatenate Numeric Features

```
python

from sklearn.feature_extraction.text import TfidfVectorizer
from scipy.sparse import hstack

# TF-IDF of review text
tfidf = TfidfVectorizer(max_features=1000, stop_words='english')
X_text = tfidf.fit_transform(df_help['Cleaned_Text'])

# Combine with numerical features
X_num = df_help[['ReviewLength', 'SummaryLength']]
X_combined = hstack([X_text, X_num])

y = df_help['HelpfulLabel']
```

Step 3: Train Model (e.g., Logistic Regression)

```
python

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix

X_train, X_test, y_train, y_test = train_test_split(X_combined, y, test_size=0.2,
random_state=42)

model = LogisticRegression(max_iter=500)
model.fit(X_train, y_train)

y_pred = model.predict(X_test)

print("📄 Classification Report:")
print(classification_report(y_test, y_pred))

# Optional: Confusion matrix
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

cm = confusion_matrix(y_test, y_pred)
sns.heatmap(pd.DataFrame(cm), annot=True, cmap="Blues", fmt='g')
plt.title("Confusion Matrix")
```



```
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

Optional Improvements

- Use additional features (Score, sentiment polarity, review age, etc.)
- Try more advanced models: RandomForest, XGBoost, etc.
- Use **cross-validation** and **grid search** for tuning

Would you like to try a regression version or include more features like sentiment and score?



```
# Filter only rows with valid denominator
df_help = df_subset[df_subset['HelpfulnessDenominator'] > 0].copy()
df_help['HelpfulnessRatio'] = df_help['HelpfulnessNumerator'] / df_help['HelpfulnessDenominator']

# Binary target
df_help['HelpfulLabel'] = df_help['HelpfulnessRatio'].apply(lambda x: 1 if x >= 0.7 else 0)

# Text length features
df_help['ReviewLength'] = df_help['Text'].apply(lambda x: len(str(x).split()))
df_help['SummaryLength'] = df_help['Summary'].apply(lambda x: len(str(x).split()))
```

```
-----
KeyError                                Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in get_loc(self, key)
    3804     try:
-> 3805         return self._engine.get_loc(casted_key)
    3806     except KeyError as err:

index.pyx in pandas._libs.index.IndexEngine.get_loc()

index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 'HelpfulnessDenominator'
```

The error you're seeing indicates that the column '**HelpfulnessDenominator**' is **not present in your df_subset** DataFrame.

To resolve this:

Confirm These Columns Exist First

Run this to inspect the column names:

```
python

print(df_subset.columns.tolist())
```

You **should see** both:

- 'HelpfulnessNumerator'
- 'HelpfulnessDenominator'

Fix Option 1: Use Original df Instead

If your `df_subset` only contains limited columns (e.g., `Text`, `Cleaned_Text`, `Score`, etc.), switch to the original `df`:

```
python

df_help = df[df['HelpfulnessDenominator'] > 0].copy()
```

Then proceed with the rest of the feature engineering using `df_help`.

Fix Option 2: Re-create df_subset With Required Columns

If you want to use a subset **but keep the helpfulness columns**, modify how you build `df_subset`:

```
python

# Keep relevant columns when subsetting
columns_to_keep = ['Text', 'Summary', 'Score', 'HelpfulnessNumerator', 'HelpfulnessDenominator',
                  'ProductId', 'UserId']
df_subset = df[columns_to_keep].iloc[:600000].copy()
```

Then rerun your cleaning pipeline (e.g., `Cleaned_Text`, etc.) after that.

Let me know if you want me to reapply cleaning and build the full helpfulness prediction pipeline on the updated `df_subset`.



```
from sklearn.feature_extraction.text import TfidfVectorizer
from scipy.sparse import hstack

# TF-IDF of review text
tfidf = TfidfVectorizer(max_features=1000, stop_words='english')
X_text = tfidf.fit_transform(df_subset['Cleaned_Text'])

# Combine with numerical features
X_num = df_help[['ReviewLength', 'SummaryLength']]
X_combined = hstack([X_text, X_num])
```

```
y = df_help['HelpfulLabel']
```

You're almost there — but there's a **mismatch** between `df_subset` and `df_help`:

❌ Problem

- `X_text` is based on `df_subset['Cleaned_Text']`
- `X_num` and `y` are based on `df_help`, which is filtered from `df` (not necessarily the same rows)

This causes **misaligned indices**, leading to inconsistent shapes or incorrect model behavior.

✅ Fix: Ensure All Features Come from the Same Filtered DataFrame

Let's recompute `df_help` with `Cleaned_Text` included, so you can use it safely for modeling.

✅ Refined Full Code

```
python

from sklearn.feature_extraction.text import TfidfVectorizer
from scipy.sparse import hstack
import pandas as pd

# 🔄 Recreate df_help from df_subset to ensure Cleaned_Text exists
df_help = df_subset.copy()
df_help = df_help[df_help['HelpfulnessDenominator'] > 0]
df_help['HelpfulnessRatio'] = df_help['HelpfulnessNumerator'] /
df_help['HelpfulnessDenominator']
df_help['HelpfulLabel'] = df_help['HelpfulnessRatio'].apply(lambda x: 1 if x >= 0.7 else 0)

# ➡ Add text length features
df_help['ReviewLength'] = df_help['Text'].apply(lambda x: len(str(x).split()))
df_help['SummaryLength'] = df_help['Summary'].apply(lambda x: len(str(x).split()))

# 🗉 TF-IDF on cleaned review text
tfidf = TfidfVectorizer(max_features=1000, stop_words='english')
X_text = tfidf.fit_transform(df_help['Cleaned_Text']) # NOW consistent with df_help

# ➕ Combine with numeric features
X_num = df_help[['ReviewLength', 'SummaryLength']]
X_combined = hstack([X_text, X_num])

# 🎯 Target variable
y = df_help['HelpfulLabel']
```

Let me know if you'd like to proceed with:

- Splitting the data
- Training a classifier (e.g., Logistic Regression, Random Forest)
- Evaluating with classification metrics or ROC curve



```
-----
KeyError                                Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in get_loc(self, key)
    3804     try:
-> 3805         return self._engine.get_loc(casted_key)
    3806     except KeyError as err:

index.pyx in pandas._libs.index.IndexEngine.get_loc()

index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 'Cleaned_Text'
```

The above exception was the direct cause of the following exception:

```
KeyError                                Traceback (most recent call last)
2 frames
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in get_loc(self, key)
    3810     ):
    3811         raise InvalidIndexError(key)
-> 3812         raise KeyError(key) from err
    3813     except TypeError:
    3814         # If we have a listlike key, _check_indexing_error will raise

KeyError: 'Cleaned_Text'
```

Thanks! The error `KeyError: 'Cleaned_Text'` indicates that the `df_help` DataFrame **does not contain** the `Cleaned_Text` column — likely because it was filtered from a different version of `df` or `df_subset` where that column didn't exist or was lost.

✅ Fix: Regenerate `df_help` from `df_subset` that already has `Cleaned_Text`

Here is the fully corrected version:

```
python

from sklearn.feature_extraction.text import TfidfVectorizer
from scipy.sparse import hstack
import pandas as pd

# ✅ Ensure df_help is built from df_subset that has 'Cleaned_Text'
df_help = df_subset[df_subset['HelpfulnessDenominator'] > 0].copy()

# ✅ Add helpfulness features
df_help['HelpfulnessRatio'] = df_help['HelpfulnessNumerator'] /
df_help['HelpfulnessDenominator']
df_help['HelpfulLabel'] = df_help['HelpfulnessRatio'].apply(lambda x: 1 if x >= 0.7 else 0)

# ✅ Add additional engineered features
```

```
df_help['ReviewLength'] = df_help['Text'].apply(lambda x: len(str(x).split()))
df_help['SummaryLength'] = df_help['Summary'].apply(lambda x: len(str(x).split()))

# ✓ TF-IDF on Cleaned_Text (should now be present in df_help)
tfidf = TfidfVectorizer(max_features=1000, stop_words='english')
X_text = tfidf.fit_transform(df_help['Cleaned_Text'])

# ✓ Combine text and numeric features
X_num = df_help[['ReviewLength', 'SummaryLength']]
X_combined = hstack([X_text, X_num])

# 🚩 Define target
y = df_help['HelpfulLabel']
```

Would you like to:

- Train a model now (e.g. logistic regression or random forest)?
- Visualize feature importance?
- Evaluate using precision, recall, F1-score?

Let me know your next step!



```
start with this one
# Avoid division by zero by replacing denominator 0 with 1
df_subset['Helpfulness_Ratio'] = df_subset['HelpfulnessNumerator'] /
df_subset['HelpfulnessDenominator'].replace(0, 1)

# Cap values at 1 (sometimes numerators > denominators due to data issues)
df_subset['Helpfulness_Ratio'] = df_subset['Helpfulness_Ratio'].clip(upper=1.0)
```

Perfect — here's a clean and robust way to compute the **helpfulness ratio**, handling divide-by-zero issues and outliers:

```
python

# ✓ Safe helpfulness ratio calculation
df_subset['Helpfulness_Ratio'] = (
    df_subset['HelpfulnessNumerator'] / df_subset['HelpfulnessDenominator'].replace(0, 1)
)

# 🚩 Cap at 1 to handle anomalies where numerator > denominator
df_subset['Helpfulness_Ratio'] = df_subset['Helpfulness_Ratio'].clip(upper=1.0)

# 🍷 Optional: preview to confirm
df_subset[['HelpfulnessNumerator', 'HelpfulnessDenominator', 'Helpfulness_Ratio']].sample(5,
random_state=1)
```

Let me know if you'd like to:

- Create a binary label from this ratio (e.g. **Helpful** vs. **Not Helpful**),
- Visualize distribution of helpfulness,
- Or proceed to model this as a classification/regression task.



```
# Avoid division by zero by replacing denominator 0 with 1
df_subset['Helpfulness_Ratio'] = df_subset['HelpfulnessNumerator'] /
df_subset['HelpfulnessDenominator'].replace(0, 1)

# Cap values at 1 (sometimes numerators > denominators due to data issues)
df_subset['Helpfulness_Ratio'] = df_subset['Helpfulness_Ratio'].clip(upper=1.0)

# ✅ Safe helpfulness ratio calculation
df_subset['Helpfulness_Ratio'] = (
    df_subset['HelpfulnessNumerator'] / df_subset['HelpfulnessDenominator'].replace(0, 1)
)

# 🚫 Cap at 1 to handle anomalies where numerator > denominator
df_subset['Helpfulness_Ratio'] = df_subset['Helpfulness_Ratio'].clip(upper=1.0)

# 🧠 Optional: preview to confirm
df_subset[['Id', 'ProductId', 'UserId', 'ProfileName', 'HelpfulnessNumerator',
            'HelpfulnessDenominator', 'Score', 'Summary', 'Cleaned_Text',
            'Helpfulness_Ratio']].sample(5, random_state=1)
```

Your code looks solid and already implements:

- ✅ Division by zero handling
- 🚫 Capping values >1
- 🧠 Useful preview of relevant columns

But you don't need to repeat the calculation steps twice. Here's a **cleaned-up version** with just one calculation block:

```
python

# ✅ Safe helpfulness ratio calculation with divide-by-zero handling
df_subset['Helpfulness_Ratio'] = (
    df_subset['HelpfulnessNumerator'] / df_subset['HelpfulnessDenominator'].replace(0, 1)
).clip(upper=1.0) # 🚫 Cap values at 1

# 🧠 Optional: preview to confirm
df_subset[['Id', 'ProductId', 'UserId', 'ProfileName', 'HelpfulnessNumerator',
            'HelpfulnessDenominator', 'Score', 'Summary', 'Cleaned_Text',
            'Helpfulness_Ratio']].sample(5, random_state=1)
```

Let me know if you'd like to:

- Bin this ratio (e.g. helpful vs. not helpful),
- Analyze top helpful reviewers/products,
- Or predict helpfulness from review content (text modeling).





Hult_SMAFBI_Team_3__A1_Team_assignment.pdf
PDF

Dataset Selection:

Choose a relevant dataset from platforms such as Kaggle from the list:

The COVID-19 Twitter DatasetLinks to an external site., which contains tweets related to COVID-19.

The Amazon Fine Food ReviewsLinks to an external site. which contains product and user information, ratings, and a plain text review. It also includes reviews from all other Amazon categories.

The Yelp DatasetLinks to an external site. which contains user reviews, business information, and ratings across various categories.

Ensure the dataset includes text data suitable for sentiment analysis, topic modeling, and other text mining techniques.

Data Collection and Cleaning:

Collaboratively clean and preprocess the dataset to ensure its quality and reliability.

Document the steps taken to clean and prepare the data.

Descriptive Data Analysis and Code Quality:

Explore various attributes of the dataset, such as user demographics, content types, and engagement metrics.

Use Python to generate descriptive statistics and create visualizations that help in understanding the data.

Ensure your code is well-commented, clear, and error-free.

Analytical Techniques:

Apply relevant analytical techniques such as sentiment analysis, topic modeling, and network analysis to uncover patterns and trends within the dataset.

Thanks for sharing the full project context. Based on the detailed analysis already present in your notebook, here are **additional and relevant analytical techniques** you could apply (including the **network analysis** you've attempted) to enhance insights from the Amazon Fine Food Reviews dataset:

✓ 1. Network Analysis (Product-User Graph)

You already attempted this—here's how to fix the error and make it



provide appropriate full code for network analysis, based on the recent pdf file

Here's a **refined full code** to perform **network analysis** using the Amazon Fine Food Reviews dataset (as used in your project). This version focuses on building a **bipartite graph** between users and products to explore relationships and centrality:

🔗 User-Product Bipartite Network Analysis

```
python

import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt

# ✅ Step 1: Filter relevant columns (ensure existence in df_subset)
df_graph = df_subset[['UserId', 'ProductId']].dropna().drop_duplicates()

# ✅ Step 2: Sample to reduce computational load
df_sampled = df_graph.sample(n=3000, random_state=42)

# ✅ Step 3: Create bipartite graph
B = nx.Graph()
B.add_nodes_from(df_sampled['UserId'], bipartite='users')
B.add_nodes_from(df_sampled['ProductId'], bipartite='products')

# Add edges from user to product
edges = list(df_sampled.itertuples(index=False, name=None))
B.add_edges_from(edges)

# ✅ Step 4: Network stats (optional)
num_users = len(set(df_sampled['UserId']))
num_products = len(set(df_sampled['ProductId']))
print(f"Users: {num_users}, Products: {num_products}, Edges: {len(B.edges())}")

# ✅ Step 5: Project bipartite graph onto product nodes to see shared reviewers
product_nodes = set(df_sampled['ProductId'])
G_product = nx.bipartite.weighted_projected_graph(B, product_nodes)

# ✅ Step 6: Visualize the product co-review graph (top 30 for clarity)
top_n = 30
top_nodes = sorted(G_product.degree, key=lambda x: x[1], reverse=True)[:top_n]
top_products = [node for node, _ in top_nodes]
G_sub = G_product.subgraph(top_products)

plt.figure(figsize=(12, 10))
pos = nx.spring_layout(G_sub, seed=42)
nx.draw_networkx_nodes(G_sub, pos, node_size=600, node_color='lightblue')
nx.draw_networkx_edges(G_sub, pos, width=1, alpha=0.6)
nx.draw_networkx_labels(G_sub, pos, font_size=8)
plt.title("🔗 Top Product Co-Review Network (Shared Users)")
plt.axis('off')
plt.show()
```

📌 Optional Extensions

- **Centrality Analysis:**

```
python

centrality = nx.degree_centrality(G_product)
top_central_products = sorted(centrality.items(), key=lambda x: x[1], reverse=True)[:5]
print("📌 Top Central Products (by shared users):")
for prod, score in top_central_products:
    print(f"{prod}: {score:.4f}")
```

- **Community Detection:**

```
python

from networkx.algorithms import community
```



```
communities = community.greedy_modularity_communities(G_product)
print(f"Detected {len(communities)} product communities.")
```

✓ Why Network Analysis Is Useful

- Reveals **user-product interaction patterns**
- Identifies **influential products** (central nodes)
- Helps detect **product communities** frequently reviewed by similar users

Let me know if you want to **focus on user-user similarity**, or compute **PageRank**, **betweenness**, or **modularity** from this graph!



is this based on the pdf file of codes?

Yes ✓ — the network analysis code I provided **aligns directly** with your recent PDF submission titled: **"Hult_SMAFBI_Team_3__A1_Team_assignment.pdf"**

Here's how it specifically connects to your documented steps:

📌 In your PDF (Assignment Scope):

"Apply relevant analytical techniques such as sentiment analysis, topic modeling, and network analysis to uncover patterns and trends within the dataset."

✓ The provided code fulfills the **network analysis** part using:

- **UserId** and **ProductId** columns from the **Amazon Fine Food Reviews** dataset (also mentioned in the report).
- A **bipartite graph** (users ↔ products) which matches the **collaborative filtering-style connections** seen in e-commerce review datasets.
- **Projection onto product nodes** to analyze relationships between products reviewed by similar users — a typical recommendation system technique.

Enhancement Suggestions (to better reflect the PDF's analysis goals):

To ensure the code reflects *insightful patterns* from your dataset, consider:

1. Adding Sentiment Context:

- Overlay sentiment score (positive/neutral/negative) to color product nodes.

2. Linking to Topic Modeling:

- Tag product nodes with dominant LDA topic (if previously computed).

3. Include Code Documentation:

- Add markdown or comment sections per step, since your PDF emphasized code clarity and documentation.

If you want, I can help you **integrate this code into your full notebook**, or create **annotated figures** to embed in your final report or presentation. Just say the word!



provide enhancement suggestions

Here are **enhancement suggestions** to improve your **network analysis** in line with your team's assignment brief (as described in the PDF), particularly focused on **text mining insights** and **visual storytelling**:

1. Add Sentiment Labels to Edges or Nodes

Why: Helps you explore whether users with similar sentiment interact with the same or different products.

How:

- Color product nodes based on average review sentiment (positive/neutral/negative).
- Use VADER or TextBlob scores already computed.

python

```
# Compute average sentiment per product
product_sentiment = df_subset.groupby('ProductId')['VADER_Score'].mean().reset_index()
product_sentiment['SentimentLabel'] = product_sentiment['VADER_Score'].apply(
    lambda x: 'positive' if x >= 0.3 else 'negative' if x <= -0.3 else 'neutral'
)

# Merge with graph nodes
color_map = {'positive': 'green', 'neutral': 'gray', 'negative': 'red'}
node_colors = [color_map.get(product_sentiment.set_index('ProductId').loc[n]['SentimentLabel'],
'blue')
                if n in product_sentiment['ProductId'].values else 'skyblue' for n in B.nodes()]
```

2. Tag Products with Dominant Topic from LDA

Why: Connects **topic modeling** with **network structure**, helping visualize shared interests across user clusters.

How:

- After LDA modeling, assign dominant topic to each product based on reviews.
- Use node color or shape to indicate topic cluster.

```
python

# Assuming dominant_topic_df contains ['ProductId', 'Dominant_Topic']
topic_colors = sns.color_palette("hls", n_colors=10).as_hex()
topic_map = dict(zip(range(10), topic_colors))

node_colors = [topic_map.get(dominant_topic_df.set_index('ProductId').loc[n]['Dominant_Topic'],
'#cccccc')
                if n in dominant_topic_df['ProductId'].values else '#cccccc' for n in B.nodes()]
```

3. Community Detection Among Users or Products

Why: Detect user communities reviewing similar products → useful for **recommendation** or **market segmentation**.

How:

- Use `networkx.algorithms.community` to detect clusters.

```
python

from networkx.algorithms.community import greedy_modularity_communities

communities = list(greedy_modularity_communities(B))
# You can then highlight nodes from each community using different colors.
```

4. Edge Weighting by Review Count or Sentiment Strength

Why: More realistic — users who write multiple reviews or highly polarized ones should have stronger ties.

How:

```
python

# Option: Use groupby count to define edge weights
edge_weights = df_subset.groupby(['UserId', 'ProductId']).size().reset_index(name='Weight')

# Then re-create the network with weights:
B_weighted = nx.Graph()
for row in edge_weights.itertuples(index=False):
    B_weighted.add_edge(row.UserId, row.ProductId, weight=row.Weight)
```

5. Visual Layout Improvements & Legends

Why: Improve interpretability for presentation/report visuals.

How:

- Use `spring_layout` or `kamada_kawai_layout` for better spacing.
- Add legend explaining node colors (e.g., sentiment, topic).
- Use `matplotlib.patches.Patch` to manually create legends.

6. Annotated Subgraphs

Why: Highlight specific regions of interest (e.g., top-reviewed or controversial products).

How:

- Extract subgraphs of top 5 products and visualize them individually.
- Annotate user nodes with review sentiment.

Would you like me to **implement a few of these enhancements in code for you** based on your current dataset (`df_subset`)?



```
import pandas as pd
import plotly.express as px

# Copy from your working DataFrame
df_helpful = df.copy()

df['Helpfulness_Ratio'] = df['HelpfulnessNumerator'] / df['HelpfulnessDenominator'].replace(0, 1)
df['Helpfulness_Ratio'] = df['Helpfulness_Ratio'].clip(upper=1.0)

# Group by user to calculate review count and avg helpfulness
user_helpfulness = df_helpful.groupby(['UserId', 'ProfileName']).agg(
    ReviewCount=('Text', 'count'),
    AvgHelpfulness=('HelpfulnessRatio', 'mean')
).reset_index()

# Create binned reviewer activity levels
bins = [1, 10, 50, 100, 200, 500, 1000, float('inf')]
labels = ['1-10', '11-50', '51-100', '101-200', '201-500', '501-1000', '1000+']
user_helpfulness['ReviewCountBin'] = pd.cut(
    user_helpfulness['ReviewCount'],
    bins=bins,
    labels=labels,
    include_lowest=True
)

# Plot with Plotly
fig = px.box(
    user_helpfulness,
    x='ReviewCountBin',
    y='AvgHelpfulness',
    color='ReviewCountBin',
    title='📦 Helpfulness Ratio by Reviewer Activity Level',
    labels={
        'ReviewCountBin': 'Number of Reviews (Binned)',
        'AvgHelpfulness': 'Average Helpfulness Ratio'
    }
)
```

```

    },
    template='plotly_white'
)

fig.update_layout(showlegend=False)
fig.show()

it is df

```

Thanks for confirming. Since you're using the **original df DataFrame**, here's the **corrected version** of your code with a few enhancements for consistency and improved clarity:

✓ Refined Code: Helpfulness Ratio by Reviewer Activity Level

```

python

import pandas as pd
import plotly.express as px

# 🧹 Clean: Avoid division by zero and cap the ratio
df['Helpfulness_Ratio'] = df['HelpfulnessNumerator'] / df['HelpfulnessDenominator'].replace(0, 1)
df['Helpfulness_Ratio'] = df['Helpfulness_Ratio'].clip(upper=1.0)

# 👤 Group by UserId to compute metrics
user_helpfulness = df.groupby(['UserId', 'ProfileName']).agg(
    ReviewCount=('Text', 'count'),
    AvgHelpfulness=('Helpfulness_Ratio', 'mean')
).reset_index()

# 📦 Bin users by activity level (review count)
bins = [1, 10, 50, 100, 200, 500, 1000, float('inf')]
labels = ['1-10', '11-50', '51-100', '101-200', '201-500', '501-1000', '1000+']
user_helpfulness['ReviewCountBin'] = pd.cut(
    user_helpfulness['ReviewCount'],
    bins=bins,
    labels=labels,
    include_lowest=True
)

# 📊 Plot: Boxplot of helpfulness ratio by review count bin
fig = px.box(
    user_helpfulness,
    x='ReviewCountBin',
    y='AvgHelpfulness',
    color='ReviewCountBin',
    title='📦 Helpfulness Ratio by Reviewer Activity Level',
    labels={
        'ReviewCountBin': 'Number of Reviews (Binned)',
        'AvgHelpfulness': 'Average Helpfulness Ratio'
    },
    template='plotly_white'
)

fig.update_layout(showlegend=False)
fig.show()

```

🔍 Enhancement Notes:

- Ensures the correct use of **Helpfulness_Ratio** by recalculating it.

- Uses **df** consistently, as per your clarification.
- Labelled bins help interpret reviewer engagement levels.
- Suitable for insights into "Are active reviewers more helpful?"

Would you like a similar breakdown **by product** instead of by user?



```
change df name to df
import pandas as pd
import plotly.express as px

# Use df_subset or define df_clean first — this uses df_subset for consistency
df_helpful = df_subset[df_subset['HelpfulnessDenominator'] > 0].copy()

# Calculate helpfulness ratio, replacing 0 denominator with 1 and clipping
df_helpful['HelpfulnessRatio'] = (
    df_helpful['HelpfulnessNumerator'] / df_helpful['HelpfulnessDenominator'].replace(0, 1)
).clip(upper=1.0)

# Compute review length in words
df_helpful['ReviewLength'] = df_helpful['Text'].astype(str).apply(lambda x: len(x.split()))

# Plot using Plotly
fig = px.scatter(
    df_helpful,
    x='ReviewLength',
    y='HelpfulnessRatio',
    opacity=0.3,
    title='📝 Review Length vs. Helpfulness Ratio',
    labels={
        'ReviewLength': 'Review Length (Word Count)',
        'HelpfulnessRatio': 'Helpfulness Ratio'
    },
    template='plotly_white'
)

# Fine-tune layout
fig.update_traces(marker=dict(size=4))
fig.update_layout(
    height=600,
    xaxis=dict(range=[0, df_helpful['ReviewLength'].quantile(0.99)]),
    yaxis=dict(range=[0, 1])
)

fig.show()
```

Here's your **refined code** with the DataFrame name changed to **df** consistently and optimized for readability and performance:

✓ Review Length vs. Helpfulness Ratio (Using df)

```
python

import pandas as pd
import plotly.express as px

# 📦 Filter valid rows where denominator > 0
df = df[df['HelpfulnessDenominator'] > 0].copy()

# 📊 Calculate helpfulness ratio safely
df['HelpfulnessRatio'] = (
    df['HelpfulnessNumerator'] / df['HelpfulnessDenominator'].replace(0, 1)
).clip(upper=1.0)

# ✍️ Compute review length (word count)
df['ReviewLength'] = df['Text'].astype(str).apply(lambda x: len(x.split()))

# 📈 Scatter plot: Review Length vs Helpfulness Ratio
fig = px.scatter(
    df,
    x='ReviewLength',
    y='HelpfulnessRatio',
    opacity=0.3,
    title='✍️ Review Length vs. Helpfulness Ratio',
    labels={
        'ReviewLength': 'Review Length (Word Count)',
        'HelpfulnessRatio': 'Helpfulness Ratio'
    },
    template='plotly_white'
)

# 🎯 Customize layout and range for better clarity
fig.update_traces(marker=dict(size=4))
fig.update_layout(
    height=600,
    xaxis=dict(range=[0, df['ReviewLength'].quantile(0.99)]),
    yaxis=dict(range=[0, 1])
)

fig.show()
```

🔧 Suggestions:

- You may bin **ReviewLength** if the scatterplot looks too dense.
- Consider adding a regression line using `plotly.express.scatter(..., trendline='ols')` if you want to analyze the relationship.

Would you like that enhancement added as well?



```
# Step 1: Install kagglehub with pandas dataset support
!pip install kagglehub[pandas-datasets]

# Step 2: Import necessary modules
import kagglehub
from kagglehub import KaggleDatasetAdapter
```

```
# Step 3: Define the path to the specific CSV file in the dataset
file_path = "Reviews.csv" # This is the main file inside the dataset

# Step 4: Load dataset using kagglehub
df = kagglehub.load_dataset(
    KaggleDatasetAdapter.PANDAS,
    "snap/amazon-fine-food-reviews", # dataset slug from Kaggle
    file_path,
)

# Step 5: Preview the data
print("First 5 records:")
print(df.head())
```

How about it visual studio code

If you want to run this code in **Visual Studio Code (VS Code)** instead of a notebook environment like Colab or Jupyter, you'll need to adjust a few things:

✅ Here's how to make it work in VS Code:

1. Install packages in your local environment

You **must** install **kagglehub** (and optionally **kaggle**) via pip:

Open your terminal (inside VS Code) and run:

```
bash

pip install kagglehub[pandas-datasets]
```

If this fails (as **kagglehub** is sometimes finicky outside of notebooks), **you can instead:**

↻ Alternative: Manually download the dataset from Kaggle

1. Go to: <https://www.kaggle.com/datasets/snap/amazon-fine-food-reviews>
2. Download and extract the ZIP file.
3. Move **Reviews.csv** into your VS Code working directory.

Then update your code like this:

↻ VS Code-compatible version (Manual Download)

```
python

import pandas as pd

# ✅ Adjust the path to where you saved the CSV file
df = pd.read_csv("Reviews.csv")

# 🔍 Preview the data
```



```
print("First 5 records:")
print(df.head())
```

💡 Optional Enhancements

You can also add this to inspect the dataset:

```
python

print("Shape:", df.shape)
print("Columns:", df.columns)
print("Missing values:\n", df.isnull().sum())
```

Summary

Option	Recommendation
kagglehub	Good in notebooks like Colab, unstable in VSCode
Manual import	Best for VS Code—download CSV and load via pandas

Would you like me to generate a `.py` starter script tailored for VS Code?