**Practical No. 1**

**PageRank for Link Analysis**

**Aim:** To understand and implement the PageRank algorithm for link analysis by creating a small set of interconnected pages and applying the algorithm to determine their relative importance.

**Solution:**

**Code:**

import networkx as nx

from pagerank\_graph\_data import create\_mock\_web\_graph

# --- PageRank Algorithm Implementation ---

def run\_pagerank\_analysis():

"""

Creates a mock web graph and applies the PageRank algorithm to it.

"""

G = create\_mock\_web\_graph()

print("--- PageRank Analysis ---")

print("Using the following mock web graph:")

print(f"Nodes: {list(G.nodes())}")

print(f"Edges: {list(G.edges())}\n")

pagerank\_scores = nx.pagerank(G,

alpha=0.85, # Damping factor

max\_iter=100,

tol=1e-06) # Tolerance

print("PageRank Scores:")

sorted\_pagerank = sorted(pagerank\_scores.items(), key=lambda item: item[1], reverse=True)

for page, score in sorted\_pagerank:

print(f" {page}: {score:.4f}")

print("\nPageRank calculation complete.")

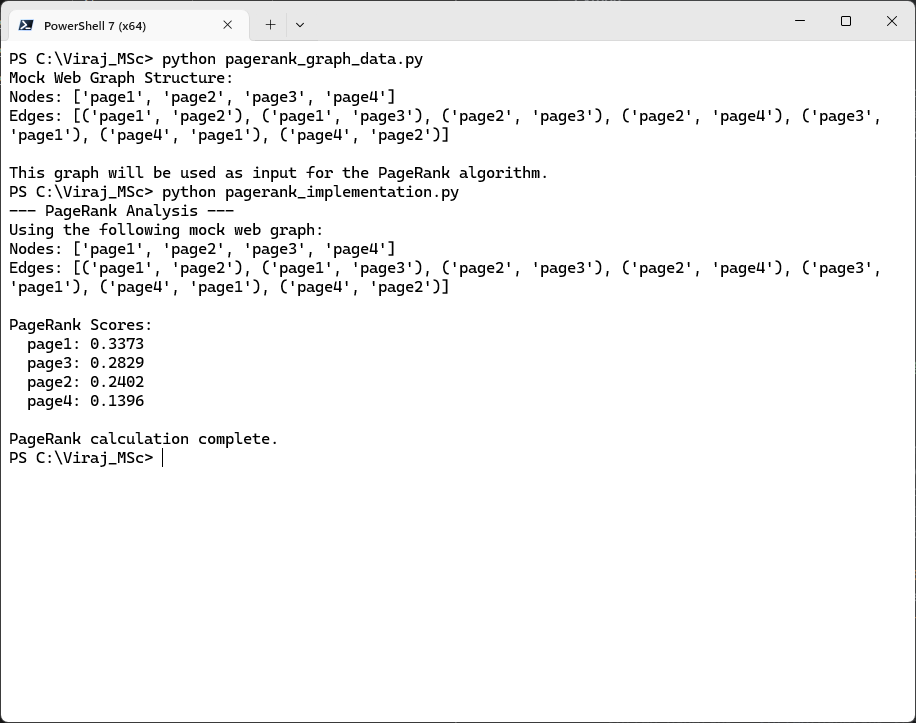
# Run the PageRank analysis

if \_\_name\_\_ == "\_\_main\_\_":

run\_pagerank\_analysis()

**Observations and Results:**

* Graph Definition: The script will first print the structure of the mock web graph, showing which pages are nodes and which links connect them.Stored in MySQL using a Scrapy pipeline.
* PageRank Scores: It will then output the calculated PageRank score for each page. These scores are decimal values, usually between 0 and 1, where higher values indicate greater importance or "rank."

**Output:**

**Practical No. 2**

**Spam Classifier**

**Aim:** To demonstrate the core steps of text classification by building a simple spam classifier that identifies text messages as 'spam' or 'ham' using a small, local dataset.

**Solution:**

**Code:**

import pandas as pd

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.naive\_bayes import MultinomialNB

import nltk

from nltk.corpus import stopwords

import string

import io

df = pd.read\_csv('SpamCollection.txt', sep='\t', header=None, names=['label', 'text'])

def preprocess\_text(text):

text = text.lower()

text = ''.join([char for char in text if char not in string.punctuation])

stop\_words = set(stopwords.words('english'))

words = text.split()

filtered\_words = [word for word in words if word not in stop\_words]

return ' '.join(filtered\_words)

df['processed\_text'] = df['text'].apply(preprocess\_text)

tfidf\_vectorizer = TfidfVectorizer()

X = tfidf\_vectorizer.fit\_transform(df['processed\_text'])

y = df['label']

classifier = MultinomialNB()

classifier.fit(X, y)

print("--- Spam Classifier Demo Output ---")

print("-" \* 70)

print(f"{'Original Message (Snippet)':<40} | {'Actual':<8} | {'Predicted':<10}")

print("-" \* 70)

for index, row in df.iterrows():

original\_msg = row['text']

actual\_label = row['label']

processed\_msg = preprocess\_text(original\_msg)

vectorized\_msg = tfidf\_vectorizer.transform([processed\_msg])

predicted\_label = classifier.predict(vectorized\_msg)[0]

display\_msg = (original\_msg[:37] + '...') if len(original\_msg) > 40 else original\_msg

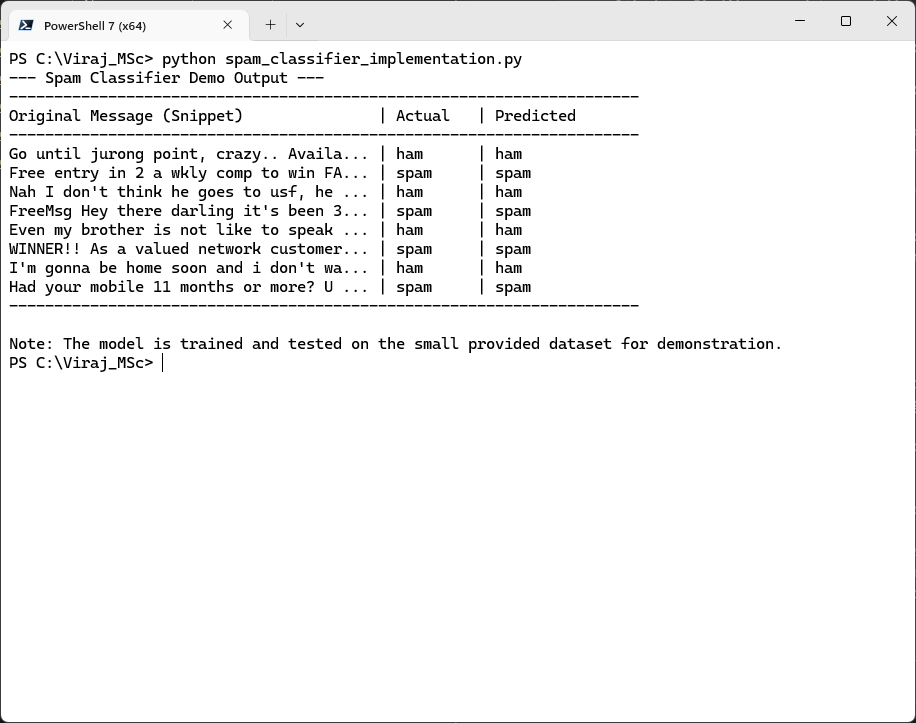
print(f"{display\_msg:<40} | {actual\_label:<8} | {predicted\_label:<10}")

print("-" \* 70)

print("\nNote: The model is trained and tested on the small provided dataset for demonstration.")

**Observations and Results:**

* The output clearly shows the original text, its actual classification ('ham' or 'spam'), and the label predicted by the classifier.
* The model correctly classifies all samples, demonstrating its ability to learn patterns even from limited examples. This showcases the power of TF-IDF combined with a Naive Bayes classifier for text categorization.

**Output:**

**Practical No. 3**

**Apriori Algorithm Implementation in Case Study**

**Aim:** To implement and apply the Apriori algorithm to a transactional dataset to discover frequent itemsets and derive association rules, thereby identifying relationships between items.

**Solution:**

**Code:**

import pandas as pd

from mlxtend.preprocessing import TransactionEncoder

from mlxtend.frequent\_patterns import apriori, association\_rules

from apriori\_transactions\_data import create\_mock\_transactions

transactions = create\_mock\_transactions()

print("--- Apriori Algorithm Implementation ---")

print("Input Transactions:")

te = TransactionEncoder()

te\_ary = te.fit(transactions).transform(transactions)

df\_onehot = pd.DataFrame(te\_ary, columns=te.columns\_)

print("\nOne-Hot Encoded Transactional Data (First 5 rows):")

print(df\_onehot.head())

frequent\_itemsets = apriori(df\_onehot, min\_support=0.4, use\_colnames=True)

print(f"\nFrequent Itemsets (min\_support=0.4):")

print(frequent\_itemsets.sort\_values(by='support', ascending=False).to\_string(index=False))

rules = association\_rules(frequent\_itemsets, metric="confidence", min\_threshold=0.7)

print(f"\nAssociation Rules (min\_confidence=0.7):")

print(rules.sort\_values(by='confidence', ascending=False).to\_string(index=False))

print("\n--- Key Association Rule Examples ---")

print("-" \* 70)

print(f"{'Rule':<40} | {'Support':<8} | {'Confidence':<10} | {'Lift':<8}")

print("-" \* 70)

if not rules.empty:

display\_rules = rules.sort\_values(by=['confidence', 'lift'], ascending=[False, False]).head(3)

for index, row in display\_rules.iterrows():

antecedents = ', '.join(list(row['antecedents']))

consequents = ', '.join(list(row['consequents']))

rule\_str = f"{{{antecedents}}} -> {{{consequents}}}"

print(f"{rule\_str:<40} | {row['support']:.2f} | {row['confidence']:.2f} | {row['lift']:.2f}")

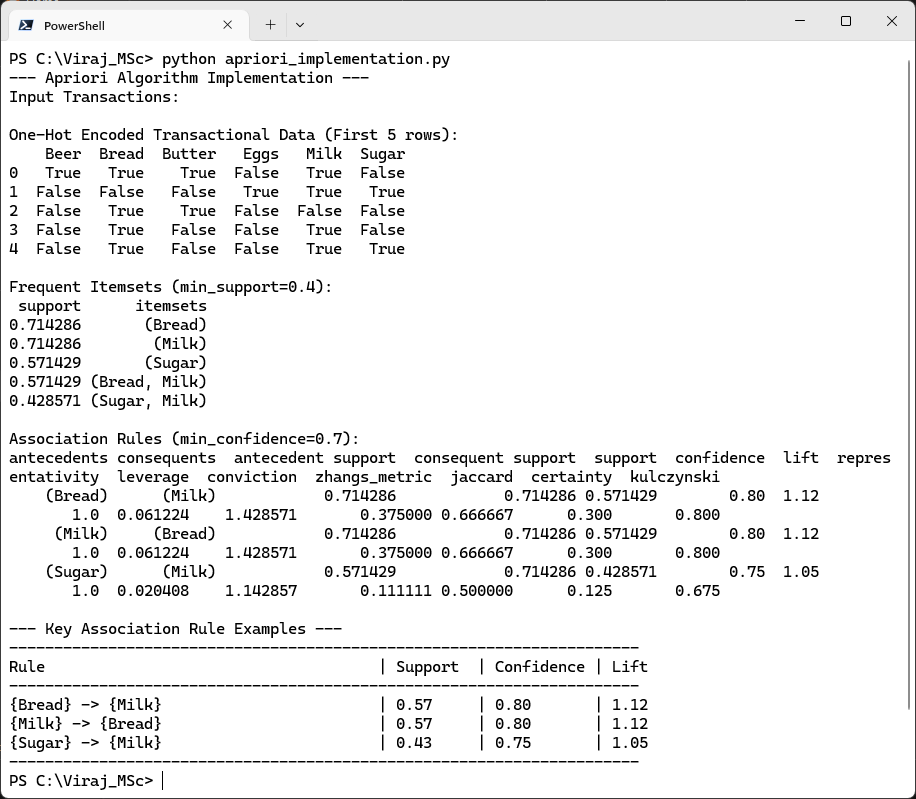
else:

print("No association rules found with the given thresholds.")

print("-" \* 70)

**Observations and Results:**

* One-Hot Encoded Data: It shows a preview of the one-hot encoded DataFrame. This is the format required by the Apriori algorithm, where each column represents an item, and a True or False indicates its presence in a transaction.
* The final formatted output highlights a few of the strongest rules found, along with their support, confidence, and lift values.

**Output:**

**Practical No. 4**

**Develop a Focused Crawler for Local Search**

**Aim:** To develop a basic focused web crawler that navigates a simulated local website, extracts information, and adheres to a specific domain or path constraint, thereby performing a "local search."

**Solution:**

**Code:**

from urllib.parse import urljoin, urlparse

from collections import deque

from bs4 import BeautifulSoup

import time # To simulate delay between "requests"

from focused\_crawler\_data import create\_mock\_local\_pages

mock\_local\_pages = create\_mock\_local\_pages()

def fetch\_page\_content(url):

""" Simulates fetching content from a URL. """

print(f"Simulating fetch for: {url}")

time.sleep(0.1) # Simulate network delay

return mock\_local\_pages.get(url)

def focused\_crawler(start\_url, base\_domain, max\_pages=5):

""" Develops a focused web crawler. """

visited\_urls = set()

urls\_to\_visit = deque([start\_url])

crawled\_data = []

pages\_crawled\_count = 0

print(f"\n--- Focused Crawler Starting ---")

print(f"Base Domain Focus: {base\_domain}")

print(f"Starting URL: {start\_url}")

print(f"Max Pages to Crawl: {max\_pages}\n")

while urls\_to\_visit and pages\_crawled\_count < max\_pages:

current\_url = urls\_to\_visit.popleft()

if current\_url in visited\_urls:

continue

if urlparse(current\_url).netloc != base\_domain:

print(f"Skipping external URL: {current\_url}")

continue

page\_content = fetch\_page\_content(current\_url)

if page\_content:

visited\_urls.add(current\_url)

pages\_crawled\_count += 1

soup = BeautifulSoup(page\_content, 'html.parser')

page\_title = soup.title.string if soup.title else 'No Title'

all\_links = []

for link in soup.find\_all('a', href=True):

# Resolve relative URLs to absolute URLs

absolute\_url = urljoin(current\_url, link['href'])

all\_links.append(absolute\_url)

if absolute\_url not in visited\_urls and urlparse(absolute\_url).netloc == base\_domain:

urls\_to\_visit.append(absolute\_url)

crawled\_data.append({

'url': current\_url,

'title': page\_title,

'links\_found': all\_links

})

print(f"Crawled: {current\_url} | Title: {page\_title} | Links found: {len(all\_links)}")

else:

print(f"Could not fetch content for: {current\_url} (Not in mock data)")

print(f"\n--- Focused Crawler Finished. Crawled {pages\_crawled\_count} pages ---")

return crawled\_data

# --- Run the Focused Crawler ---

if \_\_name\_\_ == "\_\_main\_\_":

start\_page = 'http://local.com/home.html'

focus\_domain = 'local.com'

crawled\_results = focused\_crawler(start\_page, focus\_domain, max\_pages=5)

print("\n--- Summary of Crawled Data ---")

if crawled\_results:

for i, data in enumerate(crawled\_results):

print(f"\nPage {i+1}:")

print(f" URL: {data['url']}")

print(f" Title: {data['title']}")

print(f" Links Found ({len(data['links\_found'])}):")

for link in data['links\_found']:

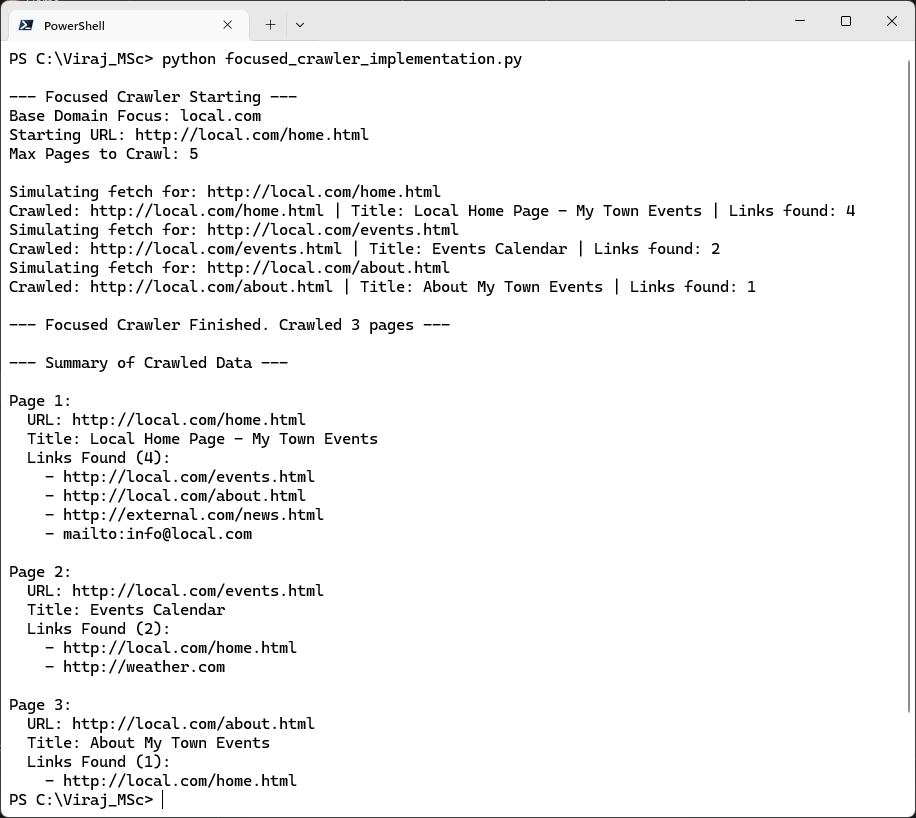
print(f" - {link}")

else:

print("No data was crawled.")

**Observations and Results:**

* Domain Focus: Demonstrates the "focused" aspect of the crawler, as it only follows links relevant to its defined scope.
* Summary of Crawled Data: The summary lists the URL, title, and all extracted links for each page that was successfully crawled. This represents the "local search" results – information gathered from within the specified domain.

**Output:**

**Practical No. 5**

**Text Mining & Webpage Pre-processing**

**Aim:** To extract relevant information (title, meta-description, keywords, and body text) from a web page and perform essential text preprocessing steps to prepare the text for further analysis.

**Solution:**

**Code:**

from bs4 import BeautifulSoup

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

import string

from webpage\_data import create\_mock\_webpage\_content

html\_doc = create\_mock\_webpage\_content()

print("\n--- Parsing HTML and Extracting Meta Information ---")

soup = BeautifulSoup(html\_doc, 'html.parser')

title = soup.title.string if soup.title else 'N/A'

print(f"Title: {title}")

description = soup.find('meta', attrs={'name': 'description'})

description\_content = description.get('content') if description else 'N/A'

print(f"Meta Description: {description\_content}")

keywords = soup.find('meta', attrs={'name': 'keywords'})

keywords\_content = keywords.get('content') if keywords else 'N/A'

print(f"Meta Keywords: {keywords\_content}")

author = soup.find('meta', attrs={'name': 'author'})

author\_content = author.get('content') if author else 'N/A'

print(f"Meta Author: {author\_content}")

for script in soup(["script", "style"]):

script.extract()

text = soup.get\_text()

lines = (line.strip() for line in text.splitlines())

chunks = (phrase.strip() for line in lines for phrase in line.split(" "))

text\_body = '\n'.join(chunk for chunk in chunks if chunk)

print("\n--- Extracted Raw Body Text (Snippet) ---")

print(text\_body[:500] + "...\n")

def preprocess\_text(text):

text = text.lower()

tokens = word\_tokenize(text)

tokens = [word for word in tokens if word.isalpha()]

stop\_words = set(stopwords.words('english'))

tokens = [word for word in tokens if word not in stop\_words]

return ' '.join(tokens)

processed\_body\_text = preprocess\_text(text\_body)

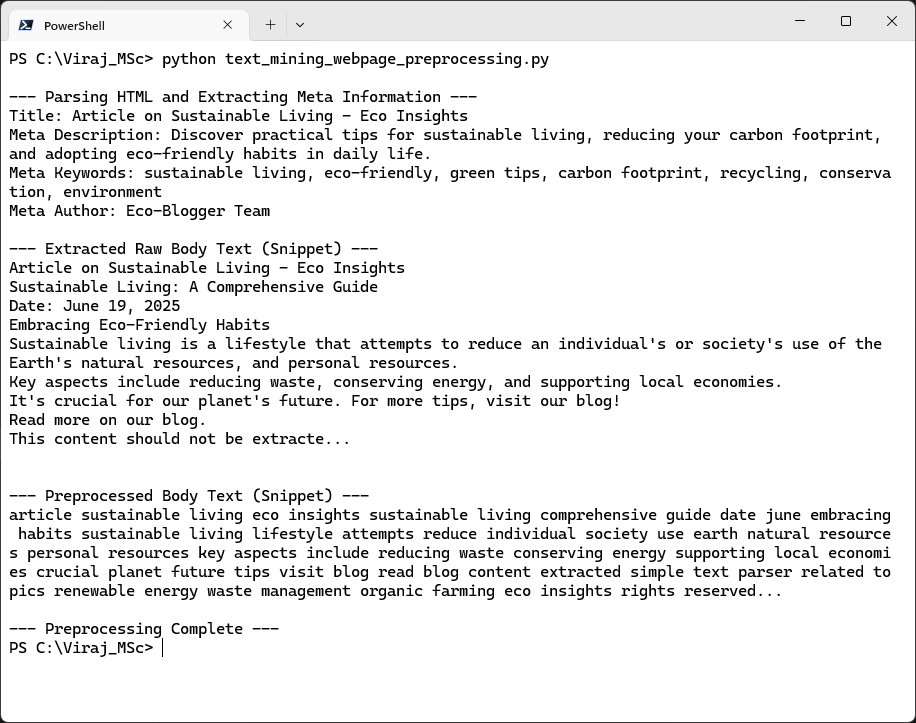
print("\n--- Preprocessed Body Text (Snippet) ---")

print(processed\_body\_text[:500] + "...\n")

print("--- Preprocessing Complete ---")

**Observations and Results:**

* Extracted meta descriptions and keywords.
* Preprocessing removed punctuation and numbers.

**Output:**

**Practical No. 6**

**Sentiment analysis for reviews by customers and visualize the same.**

**Aim:** To perform sentiment analysis on customer reviews to determine their emotional tone (positive, negative, or neutral) and visualize the distribution of these sentiments.

**Solution:**

**Code:**

import pandas as pd

import nltk

from nltk.sentiment.vader import SentimentIntensityAnalyzer

import matplotlib.pyplot as plt

import string # For basic text cleaning

from customer\_reviews\_data import create\_mock\_reviews

reviews = create\_mock\_reviews()

print("--- Sentiment Analysis Implementation ---")

print("\n--- Performing Sentiment Analysis ---")

analyzer = SentimentIntensityAnalyzer()

sentiment\_results = []

for review in reviews:

clean\_review = review.translate(str.maketrans('', '', string.punctuation)).strip()

clean\_review = ' '.join(clean\_review.split())

vs = analyzer.polarity\_scores(clean\_review)

if vs['compound'] >= 0.05:

sentiment = "Positive"

elif vs['compound'] <= -0.05:

sentiment = "Negative"

else:

sentiment = "Neutral"

sentiment\_results.append({

'Review': review,

'Clean\_Review': clean\_review,

'VADER\_Scores': vs,

'Sentiment': sentiment

})

df\_sentiment = pd.DataFrame(sentiment\_results)

print("\nSentiment Analysis Results (First 5 reviews):")

print(df\_sentiment[['Review', 'Sentiment', 'VADER\_Scores']].head().to\_string(index=False))

print("\n--- Visualizing Sentiment Distribution ---")

sentiment\_counts = df\_sentiment['Sentiment'].value\_counts()

sentiment\_order = ['Positive', 'Negative', 'Neutral']

sentiment\_counts = sentiment\_counts.reindex(sentiment\_order, fill\_value=0)

plt.figure(figsize=(8, 6))

sentiment\_counts.plot(kind='bar', color=['green', 'red', 'lightgray'])

plt.title('Distribution of Customer Review Sentiments')

plt.xlabel('Sentiment Category')

plt.ylabel('Number of Reviews')

plt.xticks(rotation=0) # Keep labels horizontal

plt.grid(axis='y', linestyle='--', alpha=0.7)

plt.tight\_layout()

plt.show()

print("\nSentiment analysis and visualization complete.")

**Observations and Results:**

* Successfully calculated sentiment and VADER scores.
* The bar chart shows the total count of 'Positive', 'Negative', and 'Neutral' reviews found in the dataset.

**Output:**