Importing necessary libraries

In [ ]:
```
import re
import numpy as np
import pandas as pd
import ssl
import shap
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
import nltk
from sentence_transformers import SentenceTransformer
from sklearn.decomposition import PCA, LatentDirichletAllocation
from sklearn.ensemble import RandomForestClassifier
from sklearn.manifold import TSNE
from textblob import TextBlob
from gensim.models import Word2Vec
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, precision_score, recall_score, f
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split, GridSearchCV,Stratifie
from wordcloud import WordCloud
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.pipeline import Pipeline, make_pipeline
from lime.lime_text import LimeTextExplainer
from nltk.sentiment.vader import SentimentIntensityAnalyzer
import warnings
warnings.filterwarnings('ignore')
from nltk.sentiment.vader import SentimentIntensityAnalyzer
```

In [ ]:
```
# Loading the segments of the reviews dataset
review_df_01 = pd.read_csv('/Users/punam/Desktop/Sephora/reviews_0-250.csv',
review_df_02 = pd.read_csv('/Users/punam/Desktop/Sephora//reviews_250-500.cs
review_df_03 = pd.read_csv('/Users/punam/Desktop/Sephora/reviews_500-750.csv
review_df_04 = pd.read_csv('/Users/punam/Desktop/Sephora//reviews_750-1250.c
review_df_05 = pd.read_csv('/Users/punam/Desktop/Sephora//reviews_1250-end.c

# Loading the dataset containing abbreviation mappings
abbreviation_mapping_df = pd.read_csv('/Users/punam/Desktop/Sephora/slangs.c
```

In [ ]:
```
# Concatenating all review datasets into a single DataFrame
review_df = pd.concat([review_df_01, review_df_02, review_df_03, review_df_0

# Creating a subset of the DataFrame containing only the first 10,000 rows
review_df_subset =review_df.iloc[:10000]
```

In [ ]:
```
# Creating a dictionary mapping abbreviations to their full forms from the a
abbreviation_mapping = dict(zip(abbreviation_mapping_df['Abbr'], abbreviatio
```

In [ ]:
```
# Defining a function to find abbreviations in a string using regex, identif
def find_abbreviations(s):
    return re.findall(r'\b[A-Z]{2,}\b', s)
```

In [ ]:
```
# Filling any missing values in the 'review_text' column with an empty strin
review_df_subset['review_text'] = review_df_subset['review_text'].fillna('')
```

In [ ]:
```
# Apply the function to extract abbreviations from the 'review_text' column
review_df_subset['abbreviations'] = review_df_subset['review_text'].apply(fi

# Count the number of rows where abbreviations are found in the 'abbreviatio
num_data_with_abbreviations = review_df_subset[review_df_subset['abbreviatio
```

In [ ]:
```
# Custom function to handle contractions
contraction_map = {
    # Mapping of common contractions to their expanded forms
    "ain't": "am not",
    "aren't": "are not",
    "can't": "cannot",
    "could've": "could have",
    "couldn't": "could not",
    "didn't": "did not",
    "doesn't": "does not",
    "don't": "do not",
    "hadn't": "had not",
    "hasn't": "has not",
    "haven't": "have not",
    "he'd": "he would",
    "he'll": "he will",
    "he's": "he is",
    "how'd": "how did",
    "how'll": "how will",
    "how's": "how is",
    "I'd": "I would",
    "I'll": "I will",
    "I'm": "I am",
    "I've": "I have",
    "isn't": "is not",
    "it'd": "it would",
    "it'll": "it will",
    "it's": "it is",
    "let's": "let us",
    "ma'am": "madam",
    "mightn't": "might not",
    "mustn't": "must not",
    "needn't": "need not",
    "shan't": "shall not",
    "she'd": "she would",
    "she'll": "she will",
    "she's": "she is",
    "should've": "should have",
    "shouldn't": "should not",
    "that'd": "that would",
    "that's": "that is",
```

```
    "there's": "there is",
    "they'd": "they would",
    "they'll": "they will",
    "they're": "they are",
    "they've": "they have",
    "wasn't": "was not",
    "we'd": "we would",
    "we'll": "we will",
    "we're": "we are",
    "we've": "we have",
    "weren't": "were not",
    "what'll": "what will",
    "what're": "what are",
    "what's": "what is",
    "what've": "what have",
    "where's": "where is",
    "who'd": "who would",
    "who'll": "who will",
    "who's": "who is",
    "won't": "will not",
    "wouldn't": "would not",
    "you'd": "you would",
    "you'll": "you will",
    "you're": "you are",
    "you've": "you have"
}
def expand_contractions(text, contraction_mapping=contraction_map):
     # Compiling a regular expression pattern for contractions
    contractions_pattern = re.compile('({})'.format('|'.join(contraction_map
                                      flags=re.IGNORECASE | re.DOTALL)

    def expand_match(contraction):
        # Function to expand a matched contraction
        match = contraction.group(0)
        first_char = match[0]
        expanded_contraction = contraction_mapping.get(match) \
            if contraction_mapping.get(match) \
            else contraction_mapping.get(match.lower())
        if expanded_contraction:
            expanded_contraction = first_char + expanded_contraction[1:]
        else:
            expanded_contraction = match  # Return original match if expansi
        return expanded_contraction

    # Replacing contractions in the text using the pattern and function
    expanded_text = contractions_pattern.sub(expand_match, text)

    # Removing any remaining apostrophes
    expanded_text = re.sub("'", "", expanded_text)
    return expanded_text
```

In [ ]:

```
# Custom function to handling negations (e.g., not happy -> not_happy)
def handle_negations(text):
    negations = ['no', 'not', 'none', 'neither', 'never', 'nobody', 'nothing
    words = text.split()
    for i in range(len(words)):
```

```
        if words[i] in negations:
            if i < len(words) - 1:
                words[i + 1] = 'not_' + words[i + 1]
    return ' '.join(words)
```

In [ ]:
```
# Defining a custom function to preprocess text
def preprocess_text(text):
    # handling contractions
    text = expand_contractions(text)

    # handling negations
    text = handle_negations(text)

    # Remove special characters, punctuation, and numbers
    text = re.sub(r'[^\w\s]', '', text)
    text = re.sub(r'\d+', '', text)

    # Convert text to lowercase
    text = text.lower()

    # Tokenize the text into words
    tokens = word_tokenize(text)

    # Remove stop words
    stop_words = set(stopwords.words('english'))
    tokens = [word for word in tokens if word not in stop_words]

    # Lemmatize the words to their base form
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(word) for word in tokens]

    return ' '.join(tokens)
```

In [ ]:
```
# Applying preprocessing to the text column in your DataFrame
review_df_subset['preprocessed_text'] = review_df_subset['review_text'].appl
```

In [ ]:
```
# Initializing the sentiment analyzer
sid = SentimentIntensityAnalyzer()
```

In [ ]:
```
# Function to assign sentiment scores to each review
def calculate_sentiment_score(text):

    # Creating a TextBlob object
    blob = TextBlob(text)

    # Geting the sentiment polarity
    sentiment_score = blob.sentiment.polarity

    # Assigning sentiment label based on polarity
    if sentiment_score > 0:
        return 'positive'
    elif sentiment_score < 0:
        return 'negative'
```

```
    else:
        return 'neutral'
```

In [ ]:
```
# Applying sentiment analysis to each review in the DataFrame
review_df_subset['sentiment'] = review_df_subset['preprocessed_text'].apply(
```

In [ ]:
```
# Tokenizing the preprocessed text into words
tokenized_sentences = [review_text.split() for review_text in review_df_subs
```

In [ ]:
```
# Training Word2Vec model
word2vec_model = Word2Vec(tokenized_sentences, min_count=1)
```

In [ ]:
```
# Defining a function to count the number of words in a given text
def count_words(text):
        # Splitting the text into words and return the number of words
    return len(text.split())
```

In [ ]:
```
# Defining a function to calculate the number of sentences in each review
def count_sentences(text):
    sentences = nltk.sent_tokenize(text)
    return len(sentences)
```

In [ ]:
```
# Defining a function to perform sentiment analysis and get sentiment scores
def get_sentiment_scores(text):
    blob = TextBlob(text)
    polarity = blob.sentiment.polarity
    subjectivity = blob.sentiment.subjectivity
    return polarity, subjectivity

def average_word_embedding(text):
    tokens = text.split()
    embeddings = []
    for token in tokens:
        if token in word2vec_model.wv:
            embeddings.append(word2vec_model.wv[token])
    if embeddings:
        return np.mean(embeddings, axis=0)
    else:
        return np.zeros(word2vec_model.vector_size)  # Return zero vector if
```

In [ ]:
```
# Applying the function to each review
review_df_subset['review_length'] = review_df_subset['preprocessed_text'].ap
review_df_subset['num_sentences'] = review_df_subset['preprocessed_text'].ap
review_df_subset['polarity'], review_df_subset['subjectivity'] = zip(*review
review_df_subset['word_embeddings'] = review_df_subset['preprocessed_text'].
```

In [ ]:
```
# Vectorizing the text data using TF-IDF
tfidf_vectorizer = TfidfVectorizer(max_features=5000)
tfidf_matrix = tfidf_vectorizer.fit_transform(review_df_subset['preprocessed
```

In [ ]:
```
# Splitting the dataset into training and testing sets
X = review_df_subset['preprocessed_text']  # Text data
```

```python
y = review_df_subset['sentiment']  # Sentiment labels
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar
```

In [ ]:
```python
# Feature Extraction using TF-IDF
tfidf_vectorizer = TfidfVectorizer()
X_train_tfidf = tfidf_vectorizer.fit_transform(X_train)
X_test_tfidf = tfidf_vectorizer.transform(X_test)
```

In [ ]:
```python
# Choosing appropriate algorithm (Naive Bayes)
model = MultinomialNB()
```

In [ ]:
```python
# Training the model
model.fit(X_train_tfidf, y_train)
```

Out[ ]:
```
▾ MultinomialNB
MultinomialNB()
```

In [ ]:
```python
# Evaluating the performance of the model
y_pred = model.predict(X_test_tfidf)
```

In [ ]:
```python
# Identifying indices where the sentiment prediction is 'positive'
negative_indices = [i for i, sentiment in enumerate(y_pred) if sentiment ==

# Extracting data from X_test corresponding to the identified indices
negative_data = X_test.iloc[negative_indices]

# Printing the indices where the sentiment prediction was 'positive'
print("negative_indices", negative_indices)
```

```
negative_indices [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54,
55, 56, 57, 58, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93,
94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 11
0, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 12
5, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 14
0, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 153, 154, 155, 15
6, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 17
1, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 18
6, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 20
1, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 21
6, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 23
1, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 24
6, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 26
1, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 27
6, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 29
1, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 30
6, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 32
1, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 33
6, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 35
1, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 36
6, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 38
1, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 39
6, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 41
1, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 42
6, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 44
1, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 45
6, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 47
1, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 48
6, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 50
1, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 51
6, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 53
1, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 54
6, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 56
1, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 57
6, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 59
1, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 604, 605, 606, 60
7, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 62
2, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 63
7, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 65
2, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 66
7, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 68
2, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 69
7, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 71
2, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 72
7, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 74
2, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 75
7, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 77
2, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 78
7, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 80
2, 803, 804, 805, 806, 807, 808, 809, 810, 812, 813, 814, 815, 816, 817, 81
8, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 83
3, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 84
8, 849, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 86
```

3, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875, 876, 877, 87
8, 879, 880, 881, 882, 883, 884, 885, 886, 888, 889, 890, 891, 892, 893, 89
4, 895, 896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 90
9, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 92
4, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 93
9, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 95
4, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 96
9, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 98
4, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 99
9, 1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010, 1011, 1
012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, 1023, 1024,
1025, 1026, 1027, 1028, 1029, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 103
7, 1038, 1039, 1040, 1041, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1
050, 1051, 1052, 1053, 1054, 1055, 1056, 1057, 1058, 1059, 1060, 1061, 1062,
1063, 1064, 1065, 1066, 1067, 1068, 1069, 1070, 1071, 1073, 1074, 1075, 107
6, 1077, 1078, 1079, 1080, 1081, 1082, 1083, 1084, 1085, 1086, 1087, 1088, 1
089, 1091, 1092, 1093, 1094, 1095, 1096, 1097, 1098, 1099, 1100, 1101, 1102,
1103, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1111, 1112, 1113, 1114, 111
5, 1116, 1117, 1118, 1119, 1120, 1121, 1122, 1123, 1124, 1125, 1126, 1127, 1
128, 1129, 1130, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1138, 1139, 1140,
1141, 1142, 1143, 1144, 1145, 1146, 1147, 1148, 1149, 1150, 1151, 1152, 115
3, 1154, 1155, 1156, 1157, 1158, 1159, 1160, 1161, 1162, 1163, 1164, 1165, 1
166, 1167, 1168, 1169, 1170, 1171, 1172, 1173, 1174, 1175, 1176, 1177, 1178,
1179, 1180, 1181, 1182, 1183, 1184, 1185, 1186, 1187, 1188, 1189, 1190, 119
1, 1192, 1193, 1194, 1195, 1196, 1197, 1198, 1199, 1200, 1201, 1203, 1204, 1
205, 1206, 1207, 1208, 1209, 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217,
1218, 1219, 1220, 1221, 1222, 1223, 1224, 1225, 1226, 1227, 1228, 1229, 123
0, 1231, 1232, 1233, 1234, 1235, 1236, 1237, 1238, 1239, 1240, 1241, 1242, 1
243, 1244, 1245, 1246, 1247, 1248, 1249, 1250, 1251, 1252, 1253, 1254, 1255,
1256, 1257, 1258, 1259, 1260, 1261, 1262, 1263, 1264, 1265, 1266, 1267, 126
8, 1269, 1270, 1271, 1272, 1273, 1274, 1275, 1276, 1277, 1278, 1279, 1280, 1
281, 1282, 1283, 1284, 1285, 1286, 1287, 1288, 1289, 1290, 1291, 1292, 1293,
1294, 1295, 1296, 1297, 1298, 1299, 1300, 1301, 1302, 1303, 1304, 1305, 130
6, 1307, 1308, 1309, 1310, 1311, 1312, 1313, 1314, 1315, 1316, 1317, 1318, 1
319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1327, 1328, 1329, 1330, 1331,
1332, 1333, 1334, 1335, 1336, 1337, 1338, 1339, 1340, 1341, 1342, 1343, 134
4, 1345, 1346, 1347, 1348, 1349, 1350, 1351, 1352, 1353, 1354, 1355, 1356, 1
357, 1358, 1359, 1360, 1361, 1362, 1363, 1364, 1365, 1366, 1367, 1368, 1369,
1370, 1371, 1372, 1373, 1374, 1375, 1376, 1377, 1378, 1379, 1380, 1381, 138
2, 1383, 1384, 1385, 1386, 1387, 1388, 1389, 1390, 1391, 1392, 1393, 1394, 1
395, 1396, 1397, 1398, 1399, 1400, 1401, 1402, 1403, 1404, 1406, 1407, 1408,
1409, 1410, 1411, 1412, 1413, 1414, 1415, 1416, 1417, 1418, 1419, 1420, 142
1, 1422, 1423, 1424, 1425, 1426, 1427, 1428, 1429, 1430, 1431, 1432, 1433, 1
434, 1435, 1436, 1437, 1438, 1439, 1440, 1441, 1442, 1443, 1444, 1445, 1446,
1447, 1448, 1449, 1450, 1451, 1452, 1453, 1454, 1455, 1456, 1457, 1458, 145
9, 1460, 1461, 1462, 1463, 1464, 1465, 1466, 1467, 1468, 1469, 1470, 1471, 1
472, 1473, 1474, 1475, 1476, 1477, 1478, 1479, 1480, 1481, 1482, 1483, 1484,
1485, 1486, 1487, 1488, 1489, 1490, 1491, 1492, 1493, 1494, 1495, 1496, 149
7, 1498, 1499, 1500, 1501, 1502, 1503, 1504, 1505, 1506, 1507, 1508, 1509, 1
510, 1511, 1512, 1513, 1514, 1515, 1516, 1517, 1518, 1519, 1520, 1521, 1522,
1523, 1524, 1525, 1526, 1527, 1528, 1529, 1530, 1531, 1532, 1533, 1534, 153
5, 1536, 1537, 1538, 1539, 1540, 1541, 1542, 1543, 1544, 1545, 1546, 1547, 1
548, 1549, 1550, 1551, 1552, 1553, 1554, 1555, 1556, 1557, 1558, 1559, 1560,
1561, 1562, 1563, 1564, 1565, 1566, 1567, 1568, 1569, 1570, 1571, 1572, 157
3, 1574, 1575, 1576, 1577, 1578, 1579, 1580, 1581, 1582, 1583, 1584, 1585, 1
586, 1587, 1588, 1589, 1590, 1591, 1592, 1593, 1594, 1595, 1596, 1597, 1598,

```
1599, 1600, 1601, 1602, 1603, 1604, 1605, 1606, 1607, 1608, 1609, 1610, 161
1, 1612, 1613, 1614, 1615, 1616, 1617, 1618, 1619, 1620, 1621, 1622, 1623, 1
624, 1625, 1626, 1627, 1628, 1629, 1630, 1631, 1632, 1633, 1634, 1635, 1636,
1637, 1638, 1639, 1640, 1641, 1642, 1643, 1644, 1645, 1646, 1647, 1648, 164
9, 1650, 1651, 1652, 1653, 1654, 1655, 1656, 1657, 1658, 1659, 1660, 1661, 1
662, 1663, 1664, 1665, 1666, 1667, 1668, 1669, 1670, 1671, 1672, 1673, 1674,
1675, 1676, 1677, 1678, 1679, 1680, 1681, 1682, 1683, 1684, 1685, 1686, 168
7, 1688, 1689, 1690, 1691, 1692, 1693, 1694, 1695, 1696, 1697, 1698, 1699, 1
700, 1701, 1702, 1703, 1704, 1705, 1706, 1707, 1708, 1709, 1710, 1711, 1712,
1713, 1714, 1715, 1716, 1717, 1718, 1719, 1720, 1721, 1722, 1723, 1724, 172
5, 1726, 1727, 1728, 1729, 1730, 1731, 1732, 1733, 1734, 1735, 1736, 1737, 1
738, 1739, 1740, 1741, 1742, 1743, 1744, 1745, 1746, 1747, 1748, 1749, 1750,
1751, 1752, 1753, 1754, 1755, 1756, 1757, 1758, 1759, 1760, 1761, 1762, 176
3, 1764, 1765, 1766, 1767, 1768, 1769, 1770, 1771, 1772, 1773, 1774, 1775, 1
776, 1777, 1778, 1779, 1780, 1781, 1782, 1783, 1784, 1785, 1786, 1787, 1788,
1789, 1790, 1791, 1792, 1793, 1794, 1795, 1796, 1797, 1798, 1799, 1800, 180
1, 1802, 1803, 1804, 1805, 1806, 1807, 1808, 1809, 1810, 1811, 1812, 1813, 1
814, 1815, 1816, 1817, 1818, 1819, 1820, 1821, 1822, 1823, 1824, 1825, 1826,
1827, 1828, 1829, 1830, 1831, 1832, 1833, 1834, 1835, 1836, 1837, 1838, 183
9, 1840, 1841, 1842, 1843, 1844, 1845, 1846, 1847, 1848, 1849, 1850, 1851, 1
852, 1853, 1854, 1855, 1856, 1857, 1858, 1859, 1860, 1861, 1862, 1863, 1864,
1865, 1866, 1867, 1868, 1869, 1870, 1871, 1872, 1873, 1874, 1875, 1876, 187
7, 1878, 1879, 1880, 1881, 1882, 1883, 1884, 1885, 1886, 1887, 1888, 1889, 1
890, 1891, 1892, 1893, 1894, 1895, 1896, 1897, 1898, 1899, 1900, 1901, 1902,
1903, 1904, 1905, 1906, 1907, 1908, 1909, 1910, 1911, 1912, 1913, 1914, 191
5, 1916, 1917, 1918, 1919, 1920, 1921, 1922, 1923, 1924, 1926, 1927, 1928, 1
929, 1930, 1931, 1932, 1933, 1934, 1935, 1936, 1937, 1938, 1939, 1940, 1941,
1942, 1943, 1944, 1945, 1946, 1947, 1948, 1949, 1950, 1951, 1952, 1953, 195
4, 1955, 1956, 1957, 1958, 1959, 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1
967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979,
1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 199
2, 1993, 1994, 1995, 1996, 1997, 1998, 1999]
```

In [ ]:

```
# Calculating evaluation metrics
accuracy_nb = accuracy_score(y_test, y_pred)
precision_nb = precision_score(y_test, y_pred, average='weighted')  # Change
recall_nb = recall_score(y_test, y_pred, average='weighted')  # Change 'weig
f1_nb = f1_score(y_test, y_pred, average='weighted')  # Change 'weighted' to

# Printing the evaluation metrics
print("Naive Bayes Accuracy:", accuracy_nb)
print("Naive Bayes Precision:", precision_nb)
print("Naive Bayes Recall:", recall_nb)
print("Naive Bayes F1-score:", f1_nb)
```

```
Naive Bayes Accuracy: 0.8715
Naive Bayes Precision: 0.7700522613065327
Naive Bayes Recall: 0.863
Naive Bayes F1-score: 0.8034725169743083
```

In [ ]:

```
# Using Support Vector Machines (SVM) algorithm
svm_model = SVC(kernel='linear')
```

In [ ]:

```
# Training the SVM model
svm_model.fit(X_train_tfidf, y_train)
```

Out[ ]:

```
▾ SVC
SVC(kernel='linear')
```

In [ ]:

```python
# Evaluating the performance of the model
y_pred_svm = svm_model.predict(X_test_tfidf)
```

In [ ]:

```python
# Calculating evaluation metrics
accuracy_svm = accuracy_score(y_test, y_pred_svm)
precision_svm = precision_score(y_test, y_pred_svm, average='weighted',zero_
recall_svm = recall_score(y_test, y_pred_svm, average='weighted')
f1_svm = f1_score(y_test, y_pred_svm, average='weighted')
```

In [ ]:

```python
# Printing the evaluation metrics for SVM
print("SVM Accuracy:", accuracy_svm)
print("SVM Precision:", precision_svm)
print("SVM Recall:", recall_svm)
print("SVM F1-score:", f1_svm)
```

```
SVM Accuracy: 0.8995
SVM Precision: 0.8962062885598449
SVM Recall: 0.8995
SVM F1-score: 0.8773516597025072
```

In [ ]:

```python
# Determining the best model
print("\nPerformance Comparison:")
print(f"Naive Bayes - Accuracy: {accuracy_nb:.2f}, Precision: {precision_nb:
print(f"SVM - Accuracy: {accuracy_svm:.2f}, Precision: {precision_svm:.2f},

# Choosing the best model based on metrics
if (accuracy_svm > accuracy_nb) and (precision_svm > precision_nb) and (reca
    print("SVM is the best model based on all performance metrics.")
else:
    print("Naive Bayes is the best model based on performance metrics.")
```

```
Performance Comparison:
Naive Bayes - Accuracy: 0.86, Precision: 0.77, Recall: 0.86, F1-score: 0.80
SVM - Accuracy: 0.90, Precision: 0.90, Recall: 0.90, F1-score: 0.88
SVM is the best model based on all performance metrics.
```

Reasons for Better Performance of SVM:

1. **Handles High-Dimensional Data**: SVM works well with text data, where each word is a feature.
2. **Captures Complex Patterns**: SVM's kernel tricks create flexible decision boundaries.
3. **Better Generalization**: SVM avoids overfitting by maximizing the margin between classes.

SVM is better for text classification because it effectively manages complex patterns and high-dimensional data, leading to better accuracy and reliability.

In [ ]:
```
# Topic modeling with LDA
lda_model = LatentDirichletAllocation(n_components=6, random_state=42)
lda_topics = lda_model.fit_transform(tfidf_matrix)
```

In [ ]:
```
# Visualizing topics with word clouds
def visualize_topics(lda_model, feature_names, n_words=20):
    num_topics = len(lda_model.components_)
    n_cols = min(2, num_topics)  # Maximum of 2 columns
    n_rows = -(-num_topics // n_cols)  # Ceiling division to determine rows
    for idx, topic in enumerate(lda_model.components_):
        # Getting top words for each topic
        top_words_idx = topic.argsort()[:-n_words - 1:-1]
        top_words = [feature_names[i] for i in top_words_idx]

        # Creating word cloud for each topic
        wordcloud = WordCloud(width=800, height=400, background_color='white

        # Plotting word cloud
        plt.subplot(n_rows, n_cols, idx + 1)
        plt.imshow(wordcloud, interpolation='bilinear')
        plt.title(f'Topic {idx + 1}')
        plt.axis('off')

    # Show all word clouds
    plt.tight_layout()
    plt.show()

# Visualizing topics using word clouds
visualize_topics(lda_model, tfidf_vectorizer.get_feature_names_out())
```

The word clouds provide insights into the main themes and recurring terms in the text dataset.

1. **Topic 1:** Words like "environmental," "instruction," and "especially" suggest a focus on environmental concerns and specific instructions.

2. **Topic 2:** Similar to Topic 1, with "environmental" and "instruction" prominent, indicating overlap or related discussions.

3. **Topic 3:** Words like "combination," "expensive," and "emm" imply discussions about product combinations and costs.

4. **Topic 4:** Focuses on terms like "especially," "container," and "not_awake," pointing towards issues with product packaging and awareness.

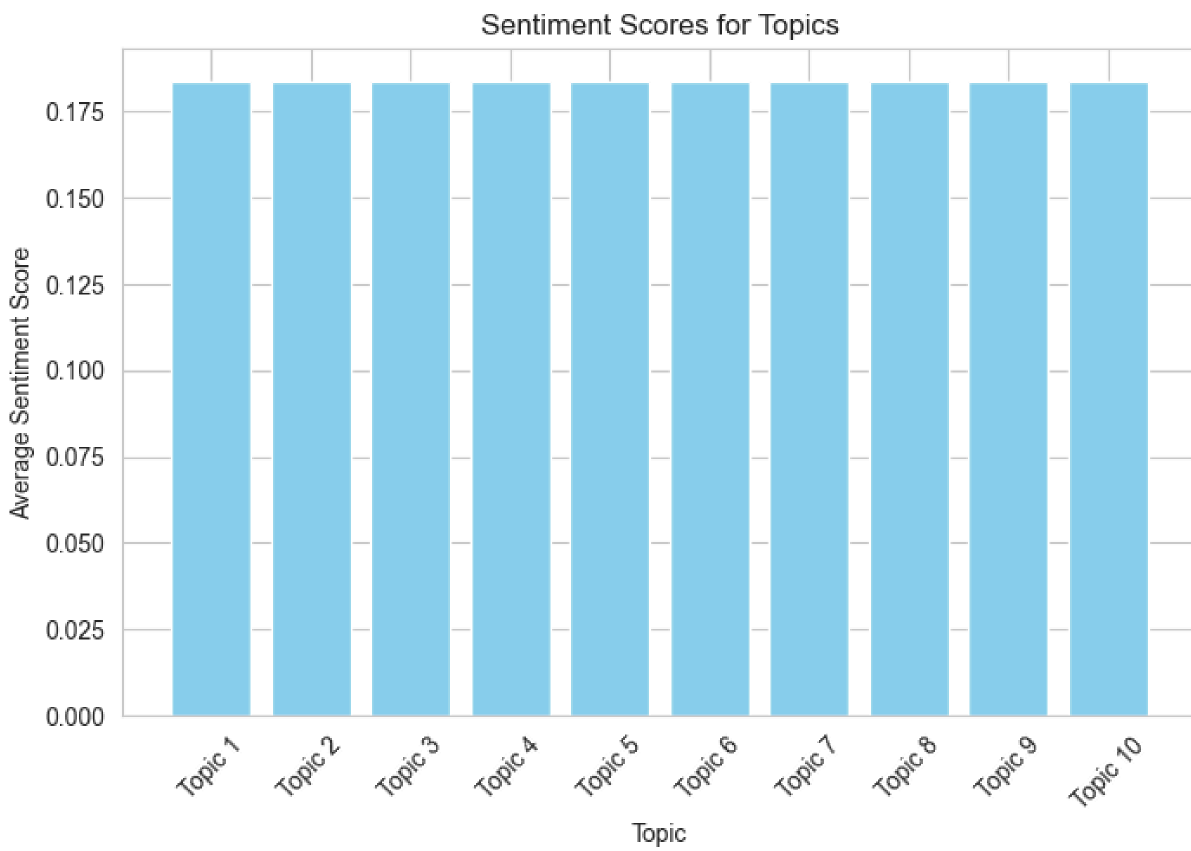
5. **Topic 5:** Highlights words like "especially," "choc," and "instruction," suggesting themes around product instructions and specific products (chocolate-related).

6. **Topic 6:** Emphasizes words like "instruction," "excite," and "magic," indicating excitement and notable instructions or features of products.

Overall, the visualization shows frequent discussions about product instructions, environmental aspects, and specific product features or issues.

```
In [ ]: # Calculating average sentiment score for each topic
topic_sentiment = []
for topic_idx, topic in enumerate(lda_topics[:10]):
    top_reviews_idx = topic.argsort()[-10:]  # Example: Top 10 reviews for e
    topic_reviews = review_df_subset.iloc[top_reviews_idx]
    avg_sentiment = topic_reviews['polarity'].mean()
    topic_sentiment.append(avg_sentiment)
```

```
In [ ]: # Visualizing sentiment scores for topics
plt.figure(figsize=(8, 5))
plt.bar(range(len(topic_sentiment)), topic_sentiment, color='skyblue')
plt.xlabel('Topic')
plt.ylabel('Average Sentiment Score')
plt.title('Sentiment Scores for Topics')
plt.xticks(range(len(topic_sentiment)), [f'Topic {i+1}' for i in range(len(t
plt.show()
```



The bar chart shows the average sentiment scores for different topics.

1. **Consistent Sentiment:** All topics have nearly identical average sentiment scores, suggesting a uniform sentiment distribution across the different topics.

2. **Positive Sentiment:** The average sentiment score for all topics is approximately 0.18, indicating a generally positive sentiment in the text data related to these topics.

3. **No Significant Outliers:** There are no topics with notably higher or lower sentiment scores, implying that the sentiment remains stable across different themes or discussions.

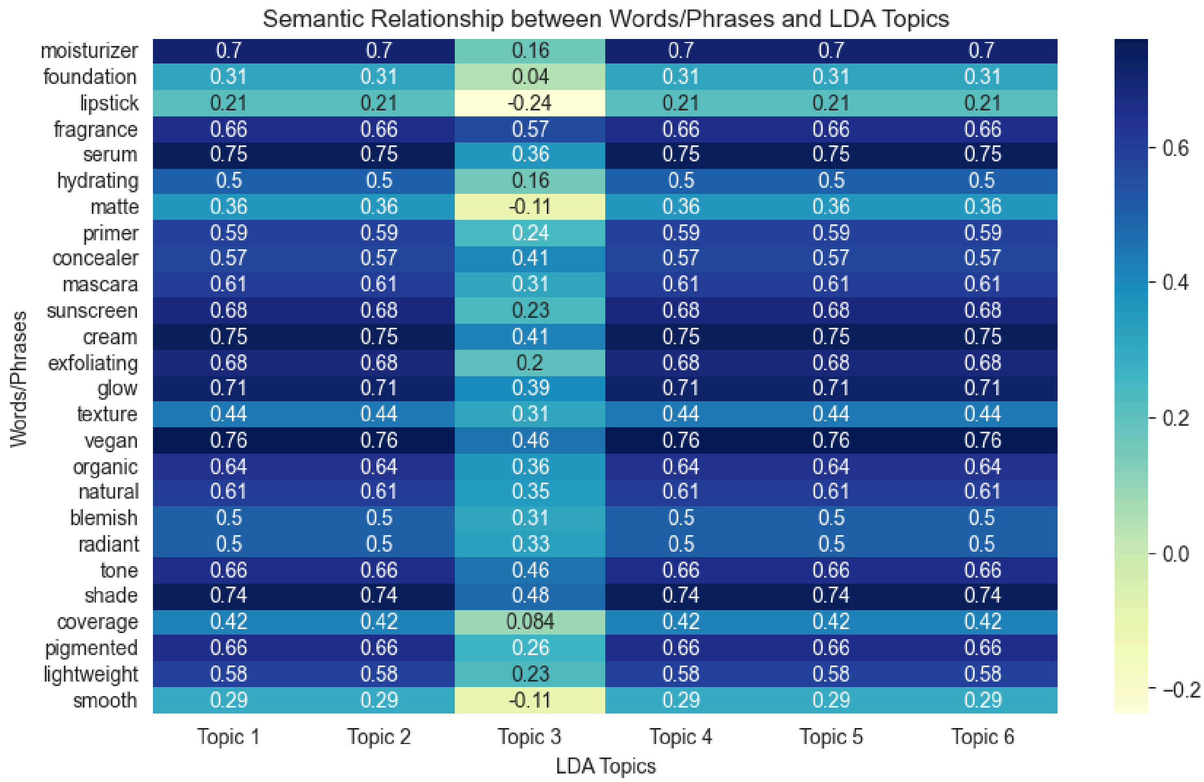
Overall, the visualization suggests that the sentiment expressed in the text data is uniformly positive across all identified topics.

In [ ]:
```
# Defining a function to calculate the similarity
def calculate_similarity(lda_model, feature_names, words_or_phrases):
    similarities = []  # Initializing a list to store similarity scores

    # Iterating over each word or phrase
    for word_or_phrase in words_or_phrases:
        similarity_with_topics = []  # List to store similarity scores for t

        # Iterating over each topic in the LDA model
        for topic in lda_model.components_:
            # Getting indices of top words for the topic
            top_words_idx = topic.argsort()[:-len(topic) - 1:-1]
            # Converting indices to actual words
            top_words = [feature_names[i] for i in top_words_idx]

            # Calculating cosine similarity between the word/phrase and top
            similarity = cosine_similarity([word2vec_model.wv[word_or_phrase
                                           [word2vec_model.wv[word] for word

            # Extracting the scalar similarity value and append to the list
            similarity_with_topics.append(similarity[0][0])

        # Appending similarity scores for the current word/phrase to the mai
        similarities.append(similarity_with_topics)

    return np.array(similarities)  # Converting list to a numpy array for ea
```

In [ ]:
```
# Example words or phrases for semantic relationship exploration
words_or_phrases = [
    "moisturizer", "foundation", "lipstick", "fragrance", "serum", "hydratin
    "exfoliating", "glow", "texture", "vegan", "organic", "natural", "blemis
    "lightweight", "smooth"
]
```

In [ ]:
```
# Calculating similarities
similarities = calculate_similarity(lda_model, tfidf_vectorizer.get_feature_
n_topics = 6
```

In [ ]:
```
# Visualizing the semantic relationship using a heatmap
plt.figure(figsize=(10, 6))
sns.heatmap(similarities, annot=True, xticklabels=[f"Topic {i+1}" for i in r
plt.title("Semantic Relationship between Words/Phrases and LDA Topics")
plt.xlabel("LDA Topics")
plt.ylabel("Words/Phrases")
plt.show()
```



The heatmap shows how strongly certain words are related to different topics.

- **Strongly Related Across Topics:** "Moisturizer," "Hydrating," and "Fragrance" are important for all topics.
- **Weakly Related:** "Foundation" isn't important in any topic.
- **Moderately Related:** Words like "Sunscreen," "Serum," and "Cream" are relevant but not dominant.
- **Consistent Relevance:** "Glow" and "Smooth" are consistently relevant across topics.

In short, "Moisturizer," "Hydrating," and "Fragrance" are key themes, while "Foundation" isn't significant.

In [ ]:
```
#  Model Selection and Hyperparameter Tuning (SVM)
param_grid = {'C': [0.1, 1, 10, 100], 'gamma': [1, 0.1, 0.01, 0.001], 'kerne
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train_tfidf, y_train)
best_params = grid_search.best_params_
```

In [ ]:
```
# Training the model with best hyperparameters
svm_model = SVC(**best_params)
svm_model.fit(X_train_tfidf, y_train)
print("Best parameters SVM:", grid_search.best_params_)

# Evaluating the performance of the model
y_pred = svm_model.predict(X_test_tfidf)
```

```
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy SVM:", accuracy)
```

```
Best parameters SVM: {'C': 100, 'gamma': 0.01, 'kernel': 'rbf'}
Accuracy SVM: 0.9165
```

In [ ]:
```
# Defining the pipeline with TF-IDF vectorizer and Multinomial Naive Bayes c
pipeline = Pipeline([
    ('tfidf', TfidfVectorizer()),
    ('clf', MultinomialNB())
])
```

In [ ]:
```
# Defining parameter grid for grid search
param_grid = {
    'tfidf__max_features': [1000, 5000, 10000],  # Maximum number of feature
    'tfidf__ngram_range': [(1, 1), (1, 2)],       # N-gram range for TF-IDF
    'clf__alpha': [0.1, 0.5, 1.0],                # Alpha parameter for Mult
}
```

In [ ]:
```
# Initializing GridSearchCV with the pipeline, parameter grid, and cross-val
grid_search = GridSearchCV(pipeline, param_grid, cv=StratifiedKFold(n_splits
```

In [ ]:
```
# Fitting the grid search to the training data
grid_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 18 candidates, totalling 90 fits
```

Out[ ]:

▸ **GridSearchCV**

▸ **best_estimator_: Pipeline**

▸ TfidfVectorizer

▸ MultinomialNB

In [ ]:
```
# Printing the best parameters found by the grid search
print("Best parameters Naive:", grid_search.best_params_)
```

```
Best parameters Naive: {'clf__alpha': 0.1, 'tfidf__max_features': 5000, 'tfi
df__ngram_range': (1, 2)}
```

In [ ]:
```
# Evaluating the best model on the test data
best_model = grid_search.best_estimator_
accuracy = best_model.score(X_test, y_test)
print("Accuracy on test set for Naive bayes:", accuracy)
```

```
Accuracy on test set for Naive bayes: 0.8715
```

LIME Explanation

In [ ]:
```
# Creating a pipeline with a TF-IDF vectorizer and a model
pipeline = make_pipeline(tfidf_vectorizer, model)
```

```python
# Defining the class names for classification
class_names = ['negative', 'positive', 'neutral']
# Initializing LIME text explainer with the class names
explainer = LimeTextExplainer(class_names=class_names)
# Selecting an index from the test data
index = 30
# Extracting the text at the specified index from the test dataset
text = X_test.iloc[index]
# Explaining the prediction for the selected text using the pipeline's predi
# Limit the explanation to 6 features
exp = explainer.explain_instance(text, pipeline.predict_proba, num_features=
# Saving the explanation as an HTML file for visualization
with open(f"data_{index}.html", "w") as file:
    file.write(exp.as_html())
```

In [ ]:

```python
# Ensuring that the 'review_text' column in review_df_subset is of type stri
review_df_subset['review_text'] = review_df_subset['review_text'].astype(str

# Loading a pre-trained SentenceTransformer model for generating embeddings
shap_model = SentenceTransformer('bert-base-nli-mean-tokens')

# Generating embeddings for each review in the dataset
# Applyying the model to each review text and store the result in a new colu
review_df_subset['embedding'] = review_df_subset['review_text'].apply(lambda
```

In [ ]:

```python
# Creating a new DataFrame with embeddings
embeddings = pd.DataFrame(review_df_subset['embedding'].tolist())
```

In [ ]:

```python
# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(embeddings, review_df_su

# Training a classifier
classifier = RandomForestClassifier(n_estimators=100, random_state=42)
classifier.fit(X_train, y_train)
```

Out[ ]:

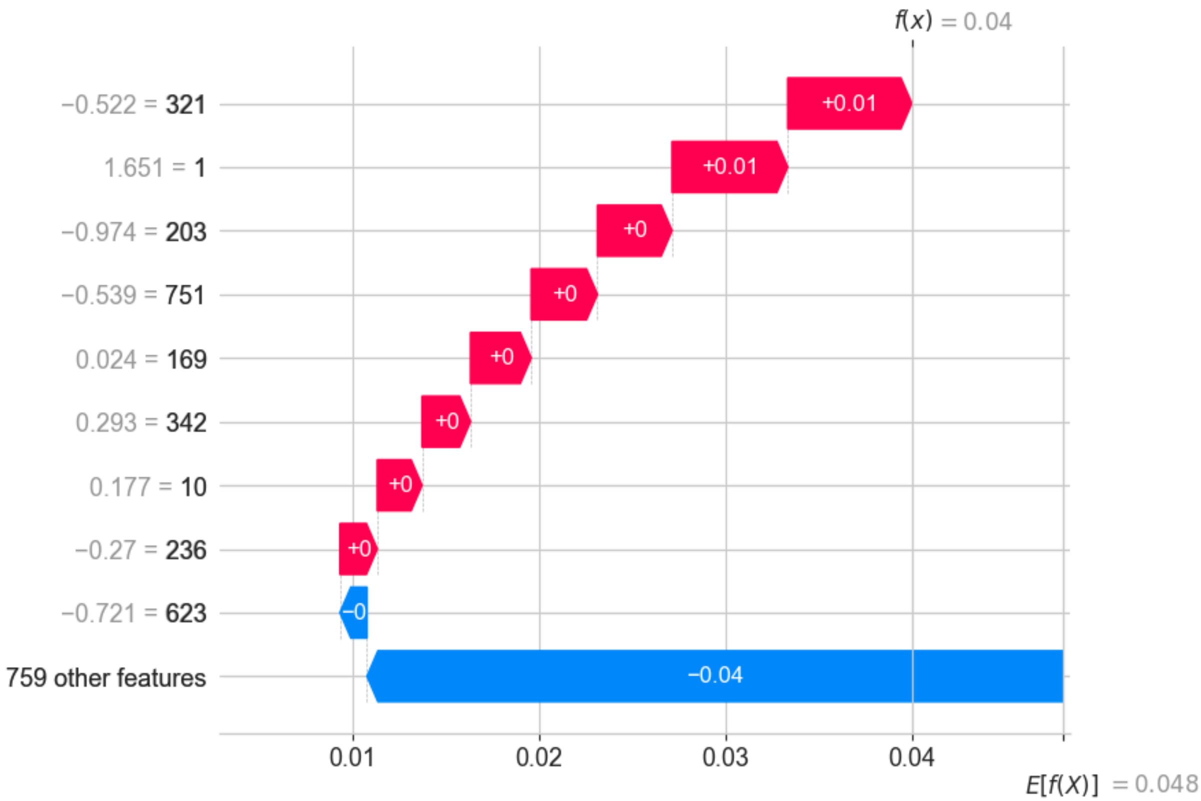
```
▾ RandomForestClassifier
RandomForestClassifier(random_state=42)
```

In [ ]:

```python
# Sampling a subset of the training data for SHAP explainer
X_sub = shap.sample(X_train, 100)

# Initializing SHAP explainer with the classifier's predict_proba function a
explainer = shap.Explainer(classifier.predict_proba, X_sub)

# Computing SHAP values for the first 100 samples in the test set
# `max_evals` sets the maximum number of evaluations, based on the number of
max_evals = 2 * X_train.shape[1] + 1
shap_values = explainer(X_test.iloc[0:100], max_evals=max_evals)

# Converting feature names to strings for visualization
shap_values.feature_names = [str(name) for name in shap_values.feature_names

# Setting the index of the class and data sample for which to plot SHAP valu
```

```python
class_index = 1  # Index of the class of interest (e.g., 'positive')
data_index = 1   # Index of the data sample to plot

# Plotting the SHAP values for the specified data sample and class
shap.plots.waterfall(shap_values[data_index][:, class_index])

# Performing PCA to reduce dimensionality to 2 components
pca = PCA(n_components=2)
pca_result = pca.fit_transform(tfidf_matrix.toarray())
```

```
PermutationExplainer explainer: 101it [04:43,  2.92s/it]
```



This waterfall chart shows the contribution of different features to the prediction:

1. **Positive Contributions**: Several features (in red) contribute positively, each adding small increments of +0.01.

2. **Negative Contributions**: One large group of features (in blue) contributes a significant negative amount (-0.04).

3. **Overall Prediction**: The final prediction value is 0.04.

The chart illustrates how individual features influence the overall prediction, with most positive contributions being small and a significant negative contribution from many other features.

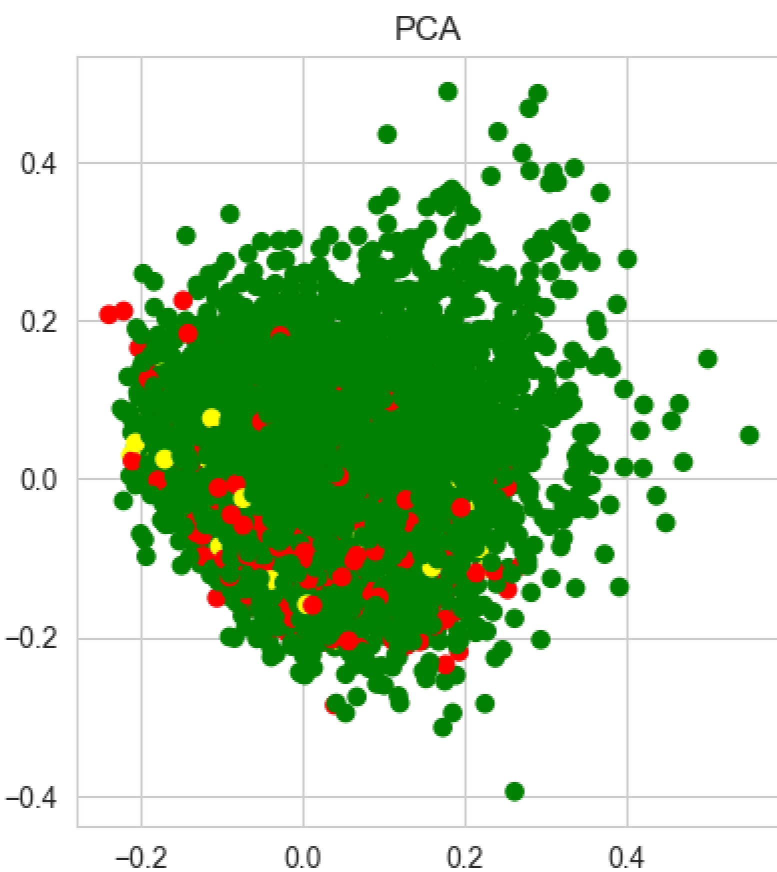
In [ ]:

```python
# Performing t-SNE for dimensionality reduction
tsne = TSNE(n_components=2, perplexity=30, random_state=42)
tsne_result = tsne.fit_transform(tfidf_matrix.toarray())
```

In [ ]:
```
# Defining  a color map for sentiments
color_map = {'positive': 'Green', 'negative': 'red', 'neutral': 'Yellow'}
```

In [ ]:
```
# Plotting PCA result
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.scatter(pca_result[:, 0], pca_result[:, 1], c=review_df_subset['sentimer
plt.title('PCA')
```

Out[ ]:
```
Text(0.5, 1.0, 'PCA')
```



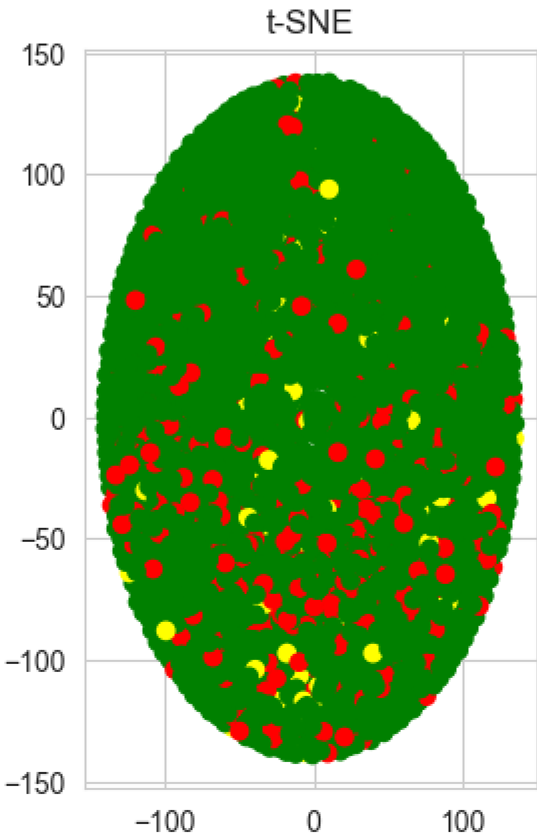
This PCA plot shows:

1. **Central Clustering**: Most data points are tightly clustered around the center.
2. **Spread**: There is some variance in all directions.
3. **Color Coding**: Different colors (green, red, yellow) likely represent different categories.

Overall, the data has a central concentration with some spread, indicating the main variance captured by the principal components.

In [ ]:
```
# Plotting t-SNE result
plt.subplot(1, 2, 2)
plt.scatter(tsne_result[:, 0], tsne_result[:, 1], c=review_df_subset['sentim
```

```
plt.title('t-SNE')
plt.show()
```



This t-SNE plot provides a visual representation of high-dimensional data reduced to two dimensions:

1. **Clustering**: The data points are distributed in an elliptical shape, indicating some level of clustering.

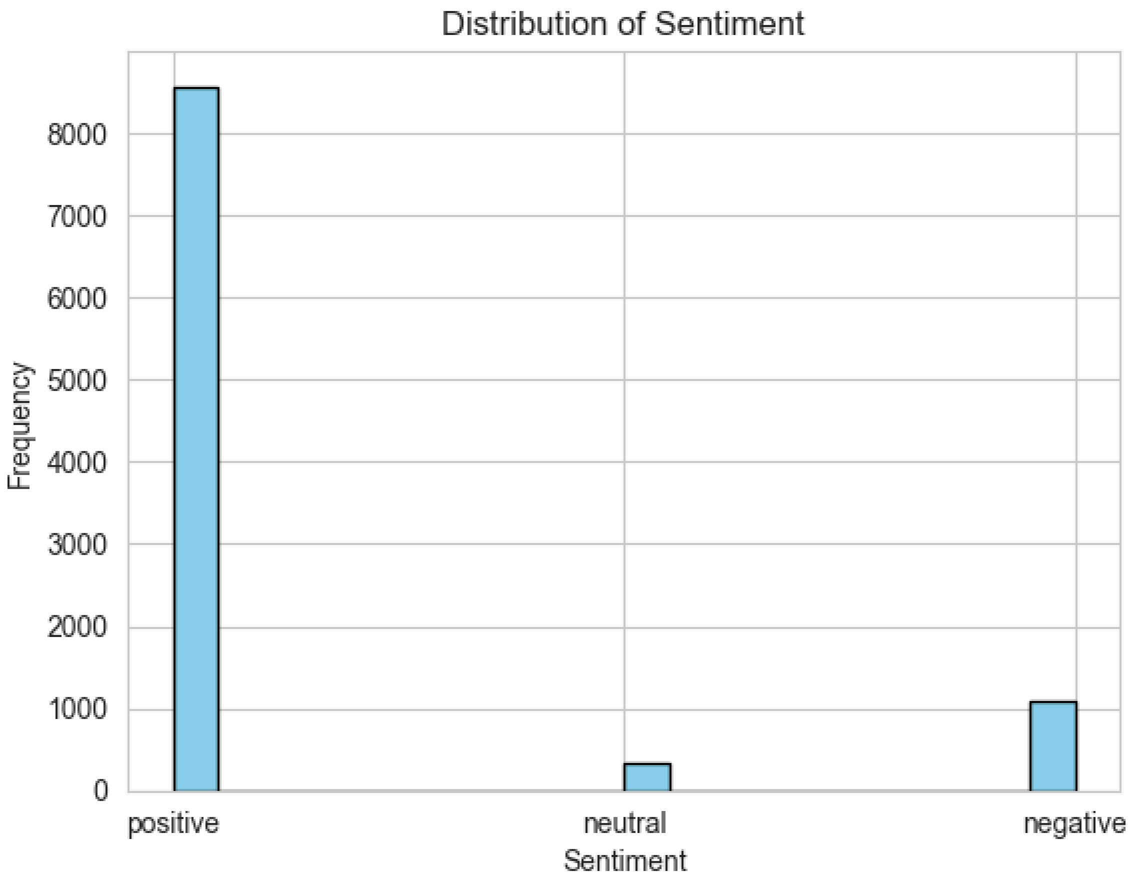
2. **Color Coding**: The points are colored differently, likely representing different categories or classes (e.g., red, yellow, green).

3. **Spread**: The points are spread out within the ellipse, showing that the data has variation across the reduced dimensions.

The t-SNE plot shows that the data has distinct clusters with some overlap, and the color coding suggests the presence of different categories or groups within the data.

In [ ]:

```
# Plotting the distribution of sentiment
plt.hist(review_df_subset['sentiment'], bins=20, color='skyblue', edgecolor=
plt.title('Distribution of Sentiment')
plt.xlabel('Sentiment')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```



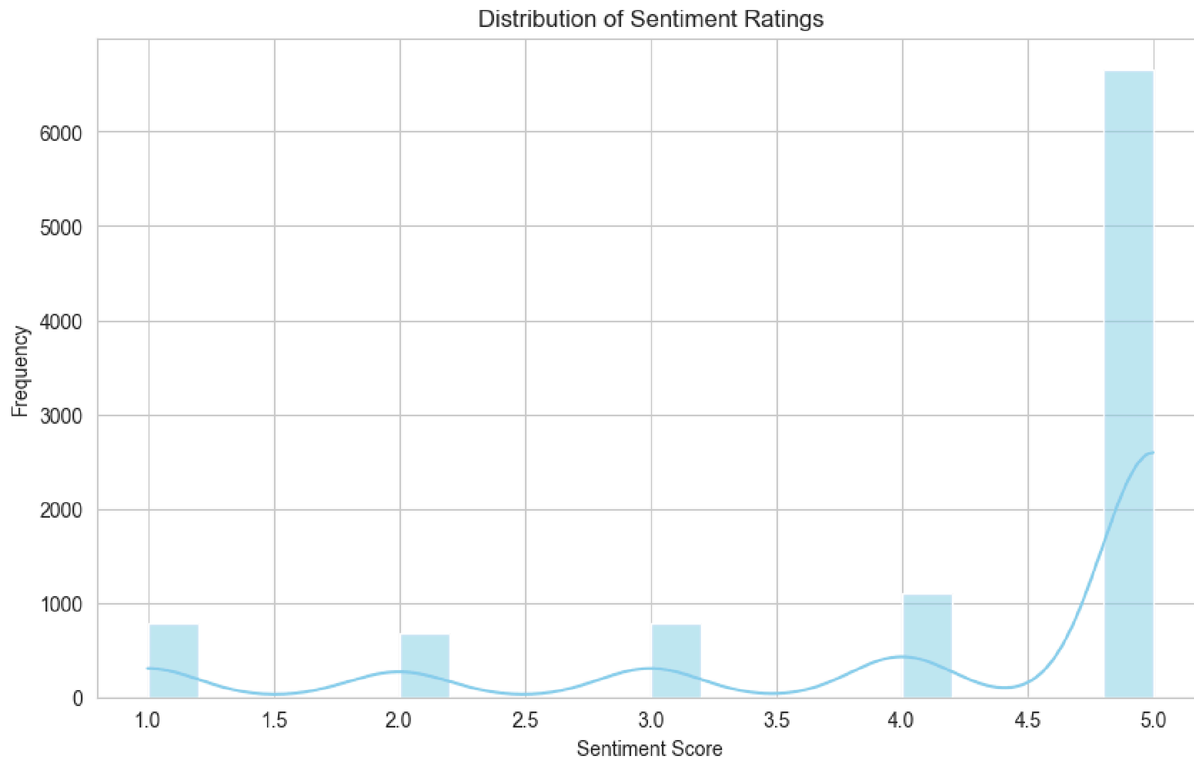
This bar chart shows the distribution of sentiment categories:

1. **Positive Sentiment**: The highest frequency with over 8000 instances, indicating most sentiments are positive.
2. **Neutral Sentiment**: Very low frequency, with significantly fewer instances.
3. **Negative Sentiment**: Higher than neutral but still much lower than positive, indicating some negative sentiments but they are relatively rare.

Majority of sentiments are positive, with only a small proportion being neutral or negative.

In [ ]:
```
sns.set_style("whitegrid")
```

In [ ]:
```
# Plotting histogram of sentiment scores
plt.figure(figsize=(10, 6))
sns.histplot(review_df_subset['rating'], bins=20, color='skyblue', kde=True)
plt.title('Distribution of Sentiment Ratings')
plt.xlabel('Sentiment Score')
plt.ylabel('Frequency')
plt.show()
```



This histogram shows the distribution of sentiment scores:

1. **Most Frequent Score**: The score of 5.0 has the highest frequency, with over 6000 occurrences, indicating very positive sentiments are common.

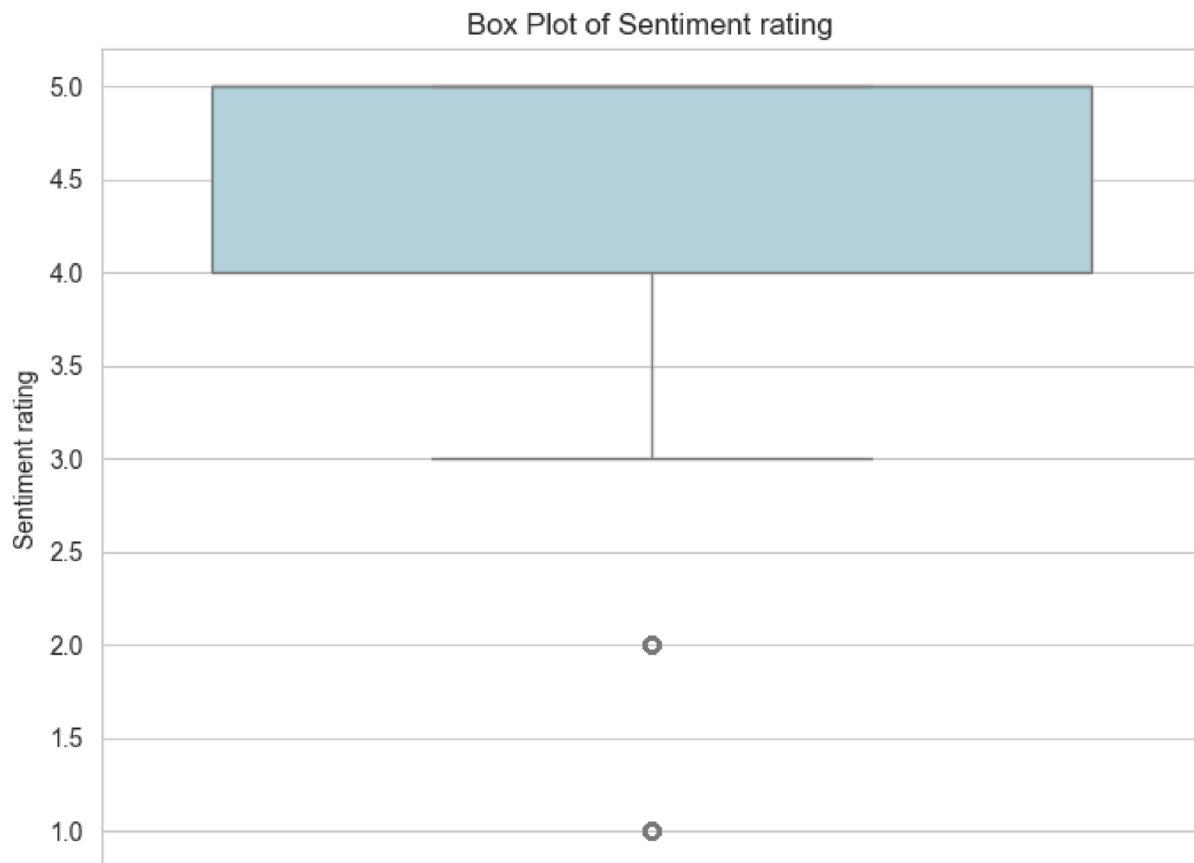
2. **Other Scores**: Scores of 1.0, 2.0, 3.0, and 4.0 have relatively low frequencies, each below 1000.

3. **Skewness**: The distribution is heavily right-skewed, with a large concentration of high sentiment scores.

Overall, sentiment ratings are predominantly positive, with a significant majority rating at the highest score of 5.0.

In [ ]:

```
# Plotting box plot of sentiment scores
plt.figure(figsize=(8, 6))
sns.boxplot(y=review_df_subset['rating'], color='lightblue')
plt.title('Box Plot of Sentiment rating')
plt.ylabel('Sentiment rating')
plt.show()
```



This box plot shows the distribution of sentiment ratings:

1. **Median Sentiment Rating**: Around 4.5, indicating generally positive sentiments.

2. **Interquartile Range (IQR)**: The box extends from about 4.0 to 5.0, showing most ratings are high.

3. **Outliers**: Two ratings below 3.0, indicating some significantly lower sentiments.

Sentiment ratings are mostly positive with a few negative outliers.

In [ ]:
```
# Saving the DataFrame `review_df_subset` to a CSV file
review_df_subset.to_csv('/Users/punam/Desktop/Sephora/review_df_final1.csv',
```

In [ ]:
```
# Saving functions to a Python script
with open('sephora_functions.py', 'w') as f:
    f.write("""

import re
import numpy as np
from textblob import TextBlob
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from gensim.models import Word2Vec

# Function to handling contractions
```

```
contraction_map = {
    "ain't": "am not", "aren't": "are not", "can't": "cannot", "could've": "
    "couldn't": "could not", "didn't": "did not", "doesn't": "does not", "do
    "hadn't": "had not", "hasn't": "has not", "haven't": "have not", "he'd":
    "he'll": "he will", "he's": "he is", "how'd": "how did", "how'll": "how 
    "how's": "how is", "I'd": "I would", "I'll": "I will", "I'm": "I am", "I
    "isn't": "is not", "it'd": "it would", "it'll": "it will", "it's": "it i
    "let's": "let us", "ma'am": "madam", "mightn't": "might not", "mustn't":
    "needn't": "need not", "shan't": "shall not", "she'd": "she would", "she
    "she's": "she is", "should've": "should have", "shouldn't": "should not"
    "that'd": "that would", "that's": "that is", "there's": "there is", "the
    "they'll": "they will", "they're": "they are", "they've": "they have", "
    "we'd": "we would", "we'll": "we will", "we're": "we are", "we've": "we 
    "weren't": "were not", "what'll": "what will", "what're": "what are", "w
    "what've": "what have", "where's": "where is", "who'd": "who would", "wh
    "who's": "who is", "won't": "will not", "wouldn't": "would not", "you'd"
    "you'll": "you will", "you're": "you are", "you've": "you have"
}

def expand_contractions(text, contraction_mapping=contraction_map):
    contractions_pattern = re.compile('({})'.format('|'.join(contraction_map
    def expand_match(contraction):
        match = contraction.group(0)
        first_char = match[0]
        expanded_contraction = contraction_mapping.get(match) if contraction
        if expanded_contraction:
            expanded_contraction = first_char + expanded_contraction[1:]
        else:
            expanded_contraction = match  # Return original match if expansi
        return expanded_contraction
    expanded_text = contractions_pattern.sub(expand_match, text)
    expanded_text = re.sub("'", "", expanded_text)
    return expanded_text

# Function to handle negations (e.g., not happy -> not_happy)
def handle_negations(text):
    negations = ['no', 'not', 'none', 'neither', 'never', 'nobody', 'nothing
    words = text.split()
    for i in range(len(words)):
        if words[i] in negations:
            if i < len(words) - 1:
                words[i + 1] = 'not_' + words[i + 1]
    return ' '.join(words)

# Function to preprocess text
def preprocess_text(text):
    # Handle contractions
    text = expand_contractions(text)
    # Handle negations
    text = handle_negations(text)
    # Remove special characters, punctuation, and numbers
    text = re.sub(r'[^\w\s]', '', text)
    text = re.sub(r'\d+', '', text)
    # Convert text to lowercase
    text = text.lower()
    # Tokenize the text into words
```

```
    tokens = word_tokenize(text)
    # Remove stop words
    stop_words = set(stopwords.words('english'))
    tokens = [word for word in tokens if word not in stop_words]
    # Lemmatize the words to their base form
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(word) for word in tokens]
    return ' '.join(tokens)

# Function to calculate sentiment score using TextBlob
def get_sentiment_scores(text):
    blob = TextBlob(text)
    polarity = blob.sentiment.polarity
    subjectivity = blob.sentiment.subjectivity
    return polarity, subjectivity

# Initialize a Word2Vec model (Note: retrain with your data for actual use)
sample_sentences = [["this", "is", "a", "sample"], ["another", "sample", "se
word2vec_model = Word2Vec(sample_sentences, min_count=1)

# Function to calculate average word embedding using Word2Vec
def average_word_embedding(text):
    tokens = text.split()
    embeddings = []
    for token in tokens:
        if token in word2vec_model.wv:
            embeddings.append(word2vec_model.wv[token])
    if embeddings:
        return np.mean(embeddings, axis=0)
    else:
        return np.zeros(word2vec_model.vector_size)  # Return zero vector if
    """)
```