

project

November 10, 2025

```
[1]: import json
import gzip
import pickle
import re
from datetime import datetime
from pathlib import Path

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from wordcloud import WordCloud

import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

from gensim import corpora
from gensim.models import LdaModel, CoherenceModel
import pyLDAvis
import pyLDAvis.gensim_models as gensimvis

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB, ComplementNB
from sklearn.metrics import (classification_report, confusion_matrix,
                             accuracy_score, f1_score,
                             precision_recall_fscore_support)
from scipy.sparse import hstack
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
import joblib

## @var OUTPUT_DIRS
# @brief Dictionary mapping output categories to their directory paths
OUTPUT_DIRS = {
```

```

'data': Path('output/data'),
'models': Path('output/models'),
'plots': Path('output/plots'),
'reports': Path('output/reports')
}

for dir_path in OUTPUT_DIRS.values():
    dir_path.mkdir(parents=True, exist_ok=True)

plt.style.use('default')
sns.set_palette("husl")

```

```
[2]: def download_nltk_data():
    """
    @brief Download required NLTK datasets for text processing

    @details Downloads stopwords, tokenizers, lemmatizers, and POS taggers
    required for natural language processing tasks.

    @return None

    @note All downloads are performed quietly to avoid verbose output
    """
    required = ['stopwords', 'punkt', 'punkt_tab', 'wordnet',
                'averaged_perceptron_tagger', 'omw-1.4']

    print("Downloading NLTK data...", end=' ')
    for item in required:
        nltk.download(item, quiet=True)
    print(" ")

def save_plot(filename, subdir='plots'):
    """
    @brief Save current matplotlib plot to output directory

    @param filename Name of file to save (with extension)
    @param subdir Subdirectory within plots folder (default: 'plots')

    @return None

    @details Saves plot with high DPI (300) and tight bounding box,
    then displays it in the notebook before closing the figure.

    @see OUTPUT_DIRS
    """

    save_path = OUTPUT_DIRS['plots'] / subdir

```

```

    save_path.mkdir(exist_ok=True)
    plt.savefig(save_path / filename, dpi=300, bbox_inches='tight')
    plt.show()
    plt.close()

def load_amazon_reviews(file_path, max_rows=None):
    """
    @brief Load Amazon reviews from compressed JSONL file

    @param file_path Path to .jsonl.gz file containing reviews
    @param max_rows Optional limit on number of reviews to load (default: None)

    @return pandas.DataFrame containing all review data

    @details Reads gzipped JSONL file line by line, parsing each JSON object
    as a review. Progress is printed every 100,000 reviews.

    @throws json.JSONDecodeError for malformed JSON lines (silently skipped)

@code{.py}
df = load_amazon_reviews('All_Beauty.jsonl.gz', max_rows=10000)
endcode
"""

reviews = []

with gzip.open(file_path, 'rt', encoding='utf-8') as f:
    for i, line in enumerate(f):
        if max_rows and i >= max_rows:
            break

        try:
            review = json.loads(line.strip())
            reviews.append(review)
        except json.JSONDecodeError:
            continue

        if (i + 1) % 100000 == 0:
            print(f"  Progress: {i + 1:,} reviews", end='\r')

print(f"\n  Loaded {len(reviews):,} reviews")
return pd.DataFrame(reviews)

def clean_text(text):
    """
    @brief Clean and normalize review text

    @param text Raw review text string

```

```

@return Cleaned text string with normalized whitespace

@details Performs the following operations:
- Converts to lowercase
- Removes HTML tags
- Removes URLs and email addresses
- Removes special characters (keeps only letters and spaces)
- Normalizes whitespace

@note Returns empty string for NaN or empty input

@code{.py}
cleaned = clean_text("<b>Great product!</b> Visit www.example.com")
# Returns: "great product visit"
endcode
"""

if pd.isna(text) or text == '':
    return ""

text = str(text).lower()
text = re.sub(r'<.*?>', '', text)
text = re.sub(r'http\S+|www.\S+', '', text)
text = re.sub(r'\S+@\S+', '', text)
text = re.sub(r'[^a-zA-Z\s]', ' ', text)
text = re.sub(r'\s+', ' ', text).strip()

return text

def preprocess_text(text):
    """
@brief Tokenize, remove stopwords, and lemmatize text

@param text Cleaned text string

@return List of processed tokens (lowercase, lemmatized)

@details Processing pipeline:
1. Clean text using clean_text()
2. Tokenize into words
3. Remove stopwords (except negation words)
4. Filter tokens shorter than 3 characters
5. Lemmatize remaining tokens

@note Preserves negation words (not, no, never, etc.) as they're
important for sentiment analysis

```

```

@see clean_text()

@code{.py}
tokens = preprocess_text("The products are not working properly!")
# Returns: ['product', 'not', 'working', 'properly']
endcode
"""

text = clean_text(text)

if not text:
    return []

tokens = word_tokenize(text)

stop_words = set(stopwords.words('english'))
keep_words = {'not', 'no', 'never', 'neither', 'nobody', 'nothing',
              'nowhere', 'hardly', 'barely', 'scarcely', "don't", "doesn't",
              "didn't", "won't", "wouldn't", "shouldn't", "couldn't", ↵
              "can't"}
stop_words = stop_words - keep_words

tokens = [w for w in tokens if len(w) > 2 and w not in stop_words]

lemmatizer = WordNetLemmatizer()
tokens = [lemmatizer.lemmatize(w) for w in tokens]

return tokens

def create_sentiment_label(rating):
    """
    @brief Convert numeric rating to sentiment category

    @param rating Numeric rating value (1-5 scale)

    @return Sentiment label: 'negative', 'neutral', or 'positive'

    @details Mapping:
    - Ratings 1-2: 'negative'
    - Rating 3: 'neutral'
    - Ratings 4-5: 'positive'

    @code{.py}
sentiment = create_sentiment_label(5) # Returns: 'positive'
sentiment = create_sentiment_label(2) # Returns: 'negative'
endcode

```

```

"""
if rating <= 2:
    return 'negative'
elif rating == 3:
    return 'neutral'
else:
    return 'positive'

def compute_coherence_values(dictionary, corpus, texts, limit=20, start=5, step=1):
    """
    @brief Compute coherence scores for different numbers of topics

    @param dictionary Gensim dictionary mapping words to IDs
    @param corpus Gensim corpus (bag-of-words representation)
    @param texts List of tokenized documents
    @param limit Maximum number of topics to test (default: 20)
    @param start Minimum number of topics to test (default: 5)
    @param step Step size for topic range (default: 1)

    @return Tuple of (model_list, coherence_values)
    @retval model_list List of trained LDA models
    @retval coherence_values List of corresponding coherence scores

    @details For each topic count in range [start, limit), trains an LDA model
    and computes its  $c_v$  coherence score. Higher coherence indicates better
    topic interpretability.

    @note This function can be computationally expensive for large corpora

    @see LdaModel, CoherenceModel

@code{.py}
models, scores = compute_coherence_values(
    dictionary, corpus, texts,
    start=5, limit=15, step=1
)
best_idx = scores.index(max(scores))
optimal_topics = list(range(5, 15))[best_idx]
@endcode
"""

coherence_values = []
model_list = []

for num_topics in range(start, limit, step):
    """
    @brief Compute coherence scores for different numbers of topics

    @param dictionary Gensim dictionary mapping words to IDs
    @param corpus Gensim corpus (bag-of-words representation)
    @param texts List of tokenized documents
    @param limit Maximum number of topics to test (default: 20)
    @param start Minimum number of topics to test (default: 5)
    @param step Step size for topic range (default: 1)

    @return Tuple of (model_list, coherence_values)
    @retval model_list List of trained LDA models
    @retval coherence_values List of corresponding coherence scores

    @details For each topic count in range [start, limit), trains an LDA model
    and computes its  $c_v$  coherence score. Higher coherence indicates better
    topic interpretability.

    @note This function can be computationally expensive for large corpora

    @see LdaModel, CoherenceModel

```

```

print(f" Testing {num_topics} topics...", end='\r')

model = LdaModel(
    corpus=corpus,
    id2word=dictionary,
    num_topics=num_topics,
    random_state=42,
    chunksize=2000,
    passes=10,
    alpha='auto',
    eta='auto',
    per_word_topics=True,
    eval_every=None
)

model_list.append(model)

coherencemodel = CoherenceModel(
    model=model,
    texts=texts,
    dictionary=dictionary,
    coherence='c_v'
)
coherence = coherencemodel.get_coherence()
coherence_values.append(coherence)

print(f"\n Tested {len(model_list)} topic configurations")
return model_list, coherence_values

def get_document_topics(lda_model, corpus):
    """
    @brief Extract topic probabilities for each document in corpus

    @param lda_model Trained LDA model
    @param corpus Gensim corpus to analyze

    @return pandas.DataFrame with topic assignments and probabilities

    @details For each document, extracts:
    - dominant_topic: Topic with highest probability
    - dominant_prob: Probability of dominant topic
    - topic_N_prob: Probability for each topic N

    @note Progress is printed every 50,000 reviews

    @code{.py}
    topics_df = get_document_topics(lda_model, corpus)
    """

```

```

print(topics_df[['dominant_topic', 'dominant_prob']].head())
endcode
"""

all_topics = []

for i, doc_topics in enumerate(lda_model[corpus]):
    if (i + 1) % 50000 == 0:
        print(f"  Processed {i + 1:,} reviews", end='\r')

    topic_probs = dict(doc_topics[0])

    if topic_probs:
        dominant_topic = max(topic_probs.items(), key=lambda x: x[1])

        topic_dict = {
            'dominant_topic': dominant_topic[0],
            'dominant_prob': dominant_topic[1]
        }

        for topic_id in range(lda_model.num_topics):
            topic_dict[f'topic_{topic_id}_prob'] = topic_probs.
get(topic_id, 0.0)

        all_topics.append(topic_dict)
    else:
        all_topics.append({
            'dominant_topic': -1,
            'dominant_prob': 0.0
        })

print("\n Topic assignment complete")
return pd.DataFrame(all_topics)

```

```

[3]: def plot_data_exploration(df):
    """
    @brief Generate comprehensive data exploration visualizations

    @param df DataFrame with review data (must contain: rating, word_count,
             timestamp, verified_purchase columns)

    @return None

    @details Creates a 2x2 subplot grid with:
    1. Rating distribution bar chart
    2. Word count histogram
    3. Reviews over time line plot

```

4. Verified vs unverified purchases pie chart

@note Saves output to 'output/plots/exploration/data_exploration.png'

@see save_plot()

"""

```
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

df['rating'].value_counts().sort_index().plot(kind='bar', ax=axes[0, 0], color='steelblue')
axes[0, 0].set_title('Rating Distribution', fontsize=14, fontweight='bold')
axes[0, 0].set_xlabel('Rating')
axes[0, 0].set_ylabel('Count')
axes[0, 0].grid(axis='y', alpha=0.3)

axes[0, 1].hist(df['word_count'], bins=50, color='coral', edgecolor='black')
axes[0, 1].set_title('Word Count Distribution', fontsize=14, fontweight='bold')
axes[0, 1].set_xlabel('Number of Words')
axes[0, 1].set_ylabel('Frequency')
axes[0, 1].set_xlim(0, 500)
axes[0, 1].grid(axis='y', alpha=0.3)

df['date'] = pd.to_datetime(df['timestamp'], unit='ms')
df['year_month'] = df['date'].dt.to_period('M')
reviews_over_time = df['year_month'].value_counts().sort_index()
reviews_over_time.plot(ax=axes[1, 0], color='green')
axes[1, 0].set_title('Reviews Over Time', fontsize=14, fontweight='bold')
axes[1, 0].set_xlabel('Date')
axes[1, 0].set_ylabel('Number of Reviews')
axes[1, 0].grid(alpha=0.3)

verified_counts = df['verified_purchase'].value_counts()
axes[1, 1].pie(verified_counts, labels=['Verified', 'Unverified'],
               autopct='%.1f%%', colors=['lightgreen', 'lightcoral'])
axes[1, 1].set_title('Verified vs Unverified Purchases', fontsize=14, fontweight='bold')

plt.tight_layout()
save_plot('data_exploration.png', 'exploration')

def plot_topic_coherence(num_topics, coherence_values):
    """
    @brief Plot coherence scores for different numbers of topics

    @param num_topics List of topic numbers tested
    """
```

```

@param coherence_values List of corresponding coherence scores

@return None

@details Creates line plot showing how coherence varies with topic count.
Helps identify optimal number of topics for LDA modeling.

@note Saves to 'output/plots/lda/topic_coherence_scores.png'

@see compute_coherence_values(), plot_topic_coherence()

@code{.py}
optimal_topics = find_optimal_topics(df, sample_size=50000)
print(f"Using {optimal_topics} topics for LDA model")
endcode
"""

plt.figure(figsize=(12, 6))
plt.plot(num_topics, coherence_values, marker='o', linewidth=2, ▾
markergsize=8)
plt.xlabel("Number of Topics", fontsize=12)
plt.ylabel("Coherence Score", fontsize=12)
plt.title("Topic Coherence Scores", fontsize=14, fontweight='bold')
plt.xticks(num_topics)
plt.grid(True, alpha=0.3)
plt.tight_layout()
save_plot('topic_coherence_scores.png', 'lda')

def plot_confusion_matrix_comparison(y_test, y_pred_before, y_pred_after):
    """
    @brief Compare confusion matrices before and after SMOTE balancing

    @param y_test True labels for test set
    @param y_pred_before Predictions before applying SMOTE
    @param y_pred_after Predictions after applying SMOTE

    @return None

    @details Creates side-by-side heatmaps comparing model performance
    before and after class balancing with SMOTE.

    @note Saves to 'output/plots/classification/confusion_matrix_comparison.png'

    @see balance_dataset()
    """

fig, axes = plt.subplots(1, 2, figsize=(16, 6))

```

```

labels = ['negative', 'neutral', 'positive']
label_names = ['Negative', 'Neutral', 'Positive']

cm_before = confusion_matrix(y_test, y_pred_before, labels=labels)
sns.heatmap(cm_before, annot=True, fmt='d', cmap='Blues', ax=axes[0],
            xticklabels=label_names, yticklabels=label_names,
            cbar_kws={'label': 'Count'})
axes[0].set_title('Before SMOTE (Imbalanced)', fontsize=14, fontweight='bold')
axes[0].set_ylabel('True Label', fontsize=12)
axes[0].set_xlabel('Predicted Label', fontsize=12)

cm_after = confusion_matrix(y_test, y_pred_after, labels=labels)
sns.heatmap(cm_after, annot=True, fmt='d', cmap='Greens', ax=axes[1],
            xticklabels=label_names, yticklabels=label_names,
            cbar_kws={'label': 'Count'})
axes[1].set_title('After SMOTE (Balanced)', fontsize=14, fontweight='bold')
axes[1].set_ylabel('True Label', fontsize=12)
axes[1].set_xlabel('Predicted Label', fontsize=12)

plt.tight_layout()
save_plot('confusion_matrix_comparison.png', 'classification')

def plot_f1_comparison(before_metrics, after_metrics):
    """
    @brief Compare F1 scores before and after SMOTE

    @param before_metrics Tuple of (precision, recall, f1, support) before SMOTE
    @param after_metrics Tuple of (precision, recall, f1, support) after SMOTE

    @return None

    @details Creates grouped bar chart comparing F1 scores across sentiment
    classes before and after applying SMOTE balancing.

    @note Saves to 'output/plots/classification/f1_score_comparison.png'

    @see balance_dataset()
    """
    fig, ax = plt.subplots(figsize=(10, 6))
    x = np.arange(3)
    width = 0.35

    bars1 = ax.bar(x - width/2, before_metrics[2], width,
                   label='Before SMOTE', color='lightcoral')
    bars2 = ax.bar(x + width/2, after_metrics[2], width,

```

```

label='After SMOTE', color='lightgreen')

ax.set_xlabel('Sentiment Class', fontsize=12)
ax.set_ylabel('F1 Score', fontsize=12)
ax.set_title('F1 Score Comparison: Before vs After SMOTE',
             fontsize=14, fontweight='bold')
ax.set_xticks(x)
ax.set_xticklabels(['Negative', 'Neutral', 'Positive'])
ax.legend()
ax.grid(axis='y', alpha=0.3)

for bars in [bars1, bars2]:
    for bar in bars:
        height = bar.get_height()
        ax.annotate(f'{height:.3f}', xy=(bar.get_x() + bar.get_width() / 2, height),
                    xytext=(0, 3),
                    textcoords="offset points",
                    ha='center', va='bottom', fontsize=9)

plt.tight_layout()
save_plot('f1_score_comparison.png', 'classification')

def plot_aspect_sentiment_distribution(df):
    """
    @brief Visualize sentiment distribution across product aspects

    @param df DataFrame with topic_label and predicted_sentiment columns

    @return None

    @details Creates two visualizations:
    1. Grouped bar chart of absolute counts
    2. Grouped bar chart of percentages
    3. Heatmap of sentiment distribution by aspect

    @note Saves to 'output/plots/aspect_analysis/' directory

    @see analyze_aspect()
    """

    fig, axes = plt.subplots(2, 1, figsize=(14, 12))

    aspect_sentiment_counts = pd.crosstab(df['topic_label'], df['predicted_sentiment'])
    aspect_sentiment_pct = pd.crosstab(df['topic_label'], df['predicted_sentiment'],
                                       normalize='columns')

```

```

normalize='index') * 100

aspect_sentiment_counts.plot(kind='bar', stacked=False, ax=axes[0],
                             color=['#d62728', '#ff7f0e', '#2ca02c'])
axes[0].set_title('Sentiment Distribution by Aspect (Counts)',
                  fontsize=14, fontweight='bold')
axes[0].set_xlabel('Product Aspect', fontsize=12)
axes[0].set_ylabel('Number of Reviews', fontsize=12)
axes[0].legend(title='Sentiment', labels=['Negative', 'Neutral', ↴
                                         'Positive'])
axes[0].tick_params(axis='x', rotation=45)
axes[0].grid(axis='y', alpha=0.3)

aspect_sentiment_pct.plot(kind='bar', stacked=False, ax=axes[1],
                          color=['#d62728', '#ff7f0e', '#2ca02c'])
axes[1].set_title('Sentiment Distribution by Aspect (%)',
                  fontsize=14, fontweight='bold')
axes[1].set_xlabel('Product Aspect', fontsize=12)
axes[1].set_ylabel('Percentage', fontsize=12)
axes[1].legend(title='Sentiment', labels=['Negative', 'Neutral', ↴
                                         'Positive'])
axes[1].tick_params(axis='x', rotation=45)
axes[1].grid(axis='y', alpha=0.3)

plt.tight_layout()
save_plot('aspect_sentiment_distribution.png', 'aspect_analysis')

plt.figure(figsize=(10, 8))
sns.heatmap(aspect_sentiment_counts, annot=True, fmt='d', cmap='RdYlGn',
            cbar_kws={'label': 'Number of Reviews'})
plt.title('Aspect-Based Sentiment Analysis Heatmap',
          fontsize=14, fontweight='bold')
plt.xlabel('Sentiment', fontsize=12)
plt.ylabel('Product Aspect', fontsize=12)
plt.tight_layout()
save_plot('absa_heatmap.png', 'aspect_analysis')

def generate_aspect_wordclouds(df, aspect_name):
    """
    @brief Generate word clouds for each sentiment within a product aspect

    @param df DataFrame with processed_text and predicted_sentiment columns
    @param aspect_name Name of the aspect to analyze

    @return None

    @details Creates three word clouds (positive, neutral, negative) showing
    """

```

most frequent words in reviews for the specified aspect.

@note Saves to 'output/plots/aspect_analysis/wordclouds/' directory

@see plot_aspect_sentiment_distribution()

"""

```
aspect_df = df[df['topic_label'] == aspect_name]

fig, axes = plt.subplots(1, 3, figsize=(18, 6))
sentiments = ['positive', 'neutral', 'negative']
colors = ['Greens', 'Blues', 'Reds']

for ax, sentiment, colormap in zip(axes, sentiments, colors):
    text_data = aspect_df[aspect_df['predicted_sentiment'] == sentiment][['processed_text']]

    if len(text_data) > 0:
        text = ' '.join(text_data.values)

        wordcloud = WordCloud(
            width=600, height=400,
            background_color='white',
            colormap=colormap,
            max_words=50,
            relative_scaling=0.5
        ).generate(text)

        ax.imshow(wordcloud, interpolation='bilinear')
        ax.set_title(f'{sentiment.capitalize()} ({len(text_data)} reviews)', fontsize=12, fontweight='bold')
    else:
        ax.text(0.5, 0.5, 'No reviews', ha='center', va='center')
        ax.set_title(f'{sentiment.capitalize()} (0 reviews)', fontsize=12, fontweight='bold')

    ax.axis('off')

fig.suptitle(f'Word Clouds: {aspect_name}', fontsize=14, fontweight='bold')
plt.tight_layout()
filename = f'wordcloud_{aspect_name.replace(" ", "_").replace("&", "and")}.lower().png'
save_plot(filename, 'aspect_analysis/wordclouds')

def plot_sentiment_trends(df, aspect_name=None):
    """
    @brief Plot sentiment trends over time for overall or specific aspect
    """
```

```

@param df DataFrame with year_month and predicted_sentiment columns
@param aspect_name Optional aspect name to filter by (default: None for
↪overall)

@return None

@details Creates time series plot showing how sentiment evolves over time,
either for all reviews or for a specific product aspect.

@note Saves to 'output/plots/aspect_analysis/trends/' directory

@see analyze_aspect()

"""

if aspect_name:
    df_plot = df[df['topic_label'] == aspect_name]
    title = f'Sentiment Trend: {aspect_name}'
    filename = f'sentiment_trend_{aspect_name.replace(" ", "_")}.
↪replace("&", "and").lower()}.png'
else:
    df_plot = df
    title = 'Overall Sentiment Trends Over Time'
    filename = 'sentiment_trend_overall.png'

sentiment_over_time = df_plot.groupby(['year_month',
↪'predicted_sentiment']).size().unstack(fill_value=0)

plt.figure(figsize=(14, 6))
sentiment_over_time.plot(kind='line', marker='o', ax=plt.gca(),
                        color=['#d62728', '#ff7f0e', '#2ca02c'])
plt.title(title, fontsize=14, fontweight='bold')
plt.xlabel('Date', fontsize=12)
plt.ylabel('Number of Reviews', fontsize=12)
plt.legend(title='Sentiment', labels=['Negative', 'Neutral', 'Positive'])
plt.grid(True, alpha=0.3)
plt.tight_layout()
save_plot(filename, 'aspect_analysis/trends')

```

```
[4]: def balance_dataset(X_train, y_train, target_positive=150000,
                      target_negative=100000, target_neutral=80000):
    """
    @brief Balance dataset using SMOTE oversampling and random undersampling

    @param X_train Training feature matrix (sparse or dense)
    @param y_train Training labels array
    @param target_positive Target count for positive class (default: 150000)

```

```

@param target_negative Target count for negative class (default: 100000)
@param target_neutral Target count for neutral class (default: 80000)

@return Tuple of (X_balanced, y_balanced)
@retval X_balanced Balanced feature matrix
@retval y_balanced Balanced labels array

@details Two-stage balancing:
1. SMOTE oversampling for minority classes (neutral, negative)
2. Random undersampling for majority class (positive)

@warning Requires imblearn package

@see naive_bayes_classification()

@code{.py}
X_bal, y_bal = balance_dataset(X_train, y_train)
print(pd.Series(y_bal).value_counts())
endcode
"""

print("\n Applying SMOTE...", end=' ')
smote = SMOTE(
    sampling_strategy={
        'neutral': target_neutral,
        'negative': target_negative
    },
    random_state=42,
    k_neighbors=5
)
X_over, y_over = smote.fit_resample(X_train, y_train)
print(" ")

print(" Applying undersampling...", end=' ')
undersample = RandomUnderSampler(
    sampling_strategy={'positive': target_positive},
    random_state=42
)
X_balanced, y_balanced = undersample.fit_resample(X_over, y_over)
print(" ")

return X_balanced, y_balanced

def train_and_evaluate_models(X_train, y_train, X_test, y_test):
"""
@brief Train and evaluate multiple Naive Bayes models

```

```

@param X_train Training feature matrix
@param y_train Training labels
@param X_test Test feature matrix
@param y_test Test labels

@return Dictionary mapping model names to results
@retval dict Keys: 'MultinomialNB', 'ComplementNB'. Each value contains:
    - 'model': trained model object
    - 'accuracy': accuracy score
    - 'f1': weighted F1 score
    - 'predictions': predicted labels

@details Trains and evaluates:
- MultinomialNB: Standard Naive Bayes for multinomial features
- ComplementNB: Better for imbalanced datasets

Both models use alpha=0.1 (Laplace smoothing parameter)

@see tune_hyperparameters()

@code{.py}
results = train_and_evaluate_models(X_train, y_train, X_test, y_test)
best_model = max(results, key=lambda k: results[k]['f1'])
@endcode
"""

models = {
    'MultinomialNB': MultinomialNB(alpha=0.1),
    'ComplementNB': ComplementNB(alpha=0.1)
}

results = []
for name, model in models.items():
    print(f" Training {name}...", end=' ')
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    acc = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred, average='weighted')

    results[name] = {
        'model': model,
        'accuracy': acc,
        'f1': f1,
        'predictions': y_pred
    }
print(f" (F1: {f1:.4f})")

```

```

    return results

def tune_hyperparameters(ModelClass, X_train, y_train):
    """
    @brief Perform grid search for optimal hyperparameters

    @param ModelClass Naive Bayes model class (MultinomialNB or ComplementNB)
    @param X_train Training feature matrix
    @param y_train Training labels

    @return Best estimator found by grid search

    @details Searches over:
    - alpha: [0.0001, 0.001, 0.01, 0.1, 0.5, 1.0, 2.0]
    - fit_prior: [True, False]

    Uses 5-fold cross-validation with weighted F1 scoring

    @note This function can be computationally expensive

    @see train_and_evaluate_models()

@code{.py}
tuned_model = tune_hyperparameters(MultinomialNB, X_train, y_train)
print(f"Best alpha: {tuned_model.alpha}")
@endcode
"""

param_grid = {
    'alpha': [0.0001, 0.001, 0.01, 0.1, 0.5, 1.0, 2.0],
    'fit_prior': [True, False]
}

print("  Performing grid search...", end=' ')
grid_search = GridSearchCV(
    ModelClass(),
    param_grid,
    cv=5,
    scoring='f1_weighted',
    n_jobs=-1,
    verbose=0
)

grid_search.fit(X_train, y_train)
print(f"  (Best F1: {grid_search.best_score_:.4f})")

return grid_search.best_estimator_

```

```

def analyze_aspect(df, aspect_name, n_examples=2):
    """
    @brief Perform detailed analysis of a specific product aspect

    @param df DataFrame with review data including topic_label,
              predicted_sentiment, rating, and text columns
    @param aspect_name Name of aspect to analyze
    @param n_examples Number of example reviews to show per sentiment (default: 2)

    @return DataFrame filtered to specified aspect

    @details Displays:
    - Total review count for aspect
    - Sentiment distribution (counts and percentages)
    - Average rating
    - Example reviews for each sentiment class

    @note Truncates review text to 200 characters in output

    @see aspect_sentiment_analysis()

@code{.py}
skin_care_df = analyze_aspect(df, "Skin Care & Sensitivity", n_examples=3)
endcode
"""

print(f"\n{'='*70}")
print(f"ASPECT: {aspect_name}")
print(f"{'='*70}")

aspect_df = df[df['topic_label'] == aspect_name].copy()

print(f"\nTotal reviews: {len(aspect_df)}")

sentiment_counts = aspect_df['predicted_sentiment'].value_counts()
print("\nSentiment Distribution:")
for sentiment in ['positive', 'neutral', 'negative']:
    if sentiment in sentiment_counts.index:
        count = sentiment_counts[sentiment]
        pct = (count / len(aspect_df)) * 100
        print(f"  {sentiment.capitalize():<10}: {count:>6,} ({pct:.1f}%)")

avg_rating = aspect_df['rating'].mean()
print(f"\nAverage Rating: {avg_rating:.2f}/5.0")

```

```

    for sentiment in ['positive', 'negative', 'neutral']:
        examples = aspect_df[aspect_df['predicted_sentiment'] == sentiment] .
        ↪head(n_examples)

        if len(examples) > 0:
            print(f"\n{sentiment.upper()} EXAMPLES:")
            for idx, (_, row) in enumerate(examples.iterrows(), 1):
                print(f"\n [{idx}] Rating: {row['rating']:.0f} | Confidence: {row['dominant_prob']:.2f}")
                text = row['text'] if len(row['text']) <= 200 else row['text'][:200] + "..."
                print(f"      \\"{text}\\"")

    return aspect_df

def save_smote_report(y_train, y_train_balanced, y_test, y_pred_before, ↪
    ↪y_pred_after):
    """
    @brief Generate and save comprehensive SMOTE improvement report

    @param y_train Original training labels (before balancing)
    @param y_train_balanced Balanced training labels (after SMOTE)
    @param y_test Test labels
    @param y_pred_before Predictions before applying SMOTE
    @param y_pred_after Predictions after applying SMOTE

    @return None

    @details Report includes:
    - Original vs balanced class distributions
    - Overall performance comparison
    - Per-class F1 and recall improvements
    - Full classification reports for both scenarios

    @note Saves to 'output/reports/smote_improvement_report.txt'

    @see balance_dataset(), plot_f1_comparison()
    """

    report_path = OUTPUT_DIRS['reports'] / 'smote_improvement_report.txt'

    before_metrics = precision_recall_fscore_support(y_test, y_pred_before,
                                                       labels=['negative', ↪
                                                               'neutral', 'positive'])
    after_metrics = precision_recall_fscore_support(y_test, y_pred_after,
                                                       labels=['negative', ↪
                                                               'neutral', 'positive'])

```

```

improvement_df = pd.DataFrame({
    'Class': ['Negative', 'Neutral', 'Positive'],
    'F1_Before': before_metrics[2],
    'F1_After': after_metrics[2],
    'F1_Improvement': after_metrics[2] - before_metrics[2],
    'Recall_Before': before_metrics[1],
    'Recall_After': after_metrics[1],
    'Recall_Improvement': after_metrics[1] - before_metrics[1]
})

with open(report_path, 'w') as f:
    f.write("=="*70 + "\n")
    f.write("SMOTE IMPROVEMENT REPORT\n")
    f.write("=="*70 + "\n\n")

    f.write("ORIGINAL DATA DISTRIBUTION:\n")
    f.write(f" Positive: {(y_train == 'positive').sum():,} ({(y_train == 'positive').mean()*100:.1f}%) \n")
    f.write(f" Negative: {(y_train == 'negative').sum():,} ({(y_train == 'negative').mean()*100:.1f}%) \n")
    f.write(f" Neutral: {(y_train == 'neutral').sum():,} ({(y_train == 'neutral').mean()*100:.1f}%) \n\n")

    f.write("BALANCED DATA DISTRIBUTION:\n")
    f.write(f" Positive: {(y_train_balanced == 'positive').sum():,}\n")
    f.write(f" Negative: {(y_train_balanced == 'negative').sum():,}\n")
    f.write(f" Neutral: {(y_train_balanced == 'neutral').sum():,}\n\n")

    f.write("PERFORMANCE COMPARISON:\n")
    f.write(f" Before SMOTE - Weighted F1: {f1_score(y_test, y_pred_before, average='weighted'):.4f}\n")
    f.write(f" After SMOTE - Weighted F1: {f1_score(y_test, y_pred_after, average='weighted'):.4f}\n\n")

    f.write("PER-CLASS F1 IMPROVEMENTS:\n")
    f.write(improvement_df.to_string(index=False))
    f.write("\n\n")

    f.write("BEFORE SMOTE:\n")
    f.write(classification_report(y_test, y_pred_before))
    f.write("\n\nAFTER SMOTE:\n")
    f.write(classification_report(y_test, y_pred_after))

print(f" Report saved: {report_path}")

```

```
[5]: def setup():
    """
    @brief Download NLTK data and verify directory structure

    @return None

    @details Initializes the analysis environment by:
    - Downloading required NLTK datasets
    - Creating output directory structure

    @note Should be called before any other pipeline functions

    @see download_nltk_data(), OUTPUT_DIRS
    """

    download_nltk_data()
    print("  Setup complete\n")

def load_and_explore(file_path='All_Beauty.jsonl.gz'):
    """
    @brief Load and perform initial exploration of review data

    @param file_path Path to compressed JSONL file (default: 'All_Beauty.jsonl.
    ↪gz')

    @return pandas.DataFrame containing loaded reviews with added features

    @details Performs:
    - Loading reviews from compressed file
    - Computing basic statistics (text length, word count)
    - Generating exploration visualizations
    - Saving raw data to CSV

    @note Saves output to 'output/data/all_beauty_reviews.csv'

    @see load_amazon_reviews(), plot_data_exploration()

    @code{.py}
    df = load_and_explore('All_Beauty.jsonl.gz')
    print(df.shape)
    @endcode
    """

    print("\nLoading reviews...")
    df = load_amazon_reviews(file_path)
```

```

df.to_csv(OUTPUT_DIRS['data'] / 'all_beauty_reviews.csv', index=False)
print(f" Saved to {OUTPUT_DIRS['data'] / 'all_beauty_reviews.csv'}")

print(f"\nDataset shape: {df.shape}")
print(f"Columns: {df.columns.tolist()}")

print("\n" + "-"*70)
print("DATASET OVERVIEW")
print("-"*70)

print(f"\nTotal reviews: {len(df)}")
print(f"Unique users: {df['user_id'].nunique()}")
print(f"Unique products: {df['parent_asin'].nunique()}")
print(f"Verified purchases: {df['verified_purchase'].sum():,}" +
      f" ({df['verified_purchase'].mean()*100:.1f}%)")

print("\nRating distribution:")
print(df['rating'].value_counts().sort_index())

df['text_length'] = df['text'].fillna('').str.len()
df['word_count'] = df['text'].fillna('').str.split().str.len()

print(f"\nAverage text length: {df['text_length'].mean():.0f} characters")
print(f"Average word count: {df['word_count'].mean():.0f} words")

print("\nGenerating visualizations...")
plot_data_exploration(df)

return df

def preprocess_data(df=None):
    """
    @brief Clean and preprocess review text

    @param df Optional DataFrame to process. If None, loads from saved CSV

    @return pandas.DataFrame with preprocessed text and sentiment labels

    @details Processing steps:
    1. Remove null text entries
    2. Clean text (lowercase, remove HTML/URLs/special chars)
    3. Tokenize and lemmatize
    4. Filter reviews with <5 tokens
    5. Create sentiment labels from ratings

    @note Saves output to 'output/data/all_beauty_preprocessed.csv'
    """

```

```

@see clean_text(), preprocess_text(), create_sentiment_label()

@code{.py}
df = preprocess_data()
print(df['sentiment'].value_counts())
endcode
"""

if df is None:
    print("\nLoading data...")
    df = pd.read_csv(OUTPUT_DIRS['data'] / 'all_beauty_reviews.csv')

print(f"\nOriginal dataset: {len(df)} reviews")

df = df[df['text'].notna()].copy()
print(f"After removing null text: {len(df)} reviews")

print("\nCleaning text...")
df['cleaned_text'] = df['text'].apply(clean_text)

print("Tokenizing and lemmatizing...")
df['tokens'] = df['cleaned_text'].apply(preprocess_text)
df['processed_text'] = df['tokens'].apply(lambda x: ' '.join(x))

df['token_count'] = df['tokens'].apply(len)
df = df[df['token_count'] >= 5].copy()

print(f"After filtering short reviews: {len(df)} reviews")

df['sentiment'] = df['rating'].apply(create_sentiment_label)

df.to_csv(OUTPUT_DIRS['data'] / 'all_beauty_preprocessed.csv', index=False)

print("\n" + "-"*70)
print("PREPROCESSING COMPLETE")
print("-"*70)
print(f"Final dataset: {len(df)} reviews")
print("\nSentiment distribution:")
print(df['sentiment'].value_counts())

print(f"\n Saved to {OUTPUT_DIRS['data'] / 'all_beauty_preprocessed.csv'}")

return df

```

```

def find_optimal_topics(df=None, sample_size=100000):
    """
    @brief Find optimal number of topics using coherence scores

    @param df Optional DataFrame with preprocessed tokens. If None, loads from saved CSV
    @param sample_size Number of reviews to sample for coherence testing (default: 100000)

    @return int Optimal number of topics

    @details
    Performs model selection for topic modeling using LDA:
    - Creates a Gensim dictionary and Bag-of-Words corpus
    - Filters rare and overly common words
    - Trains multiple LDA models with varying topic counts
    - Evaluates coherence scores for each model
    - Selects and saves the optimal topic count

    @note
    Saves dictionary, corpus, and coherence results to `output/models/`

    @warning
    This step can be computationally expensive. Use a smaller sample size for faster testing.

    @see compute_coherence_values(), plot_topic_coherence()

    @code{.py}
optimal_topics = find_optimal_topics(df, sample_size=50000)
print(f"Optimal topics: {optimal_topics}")
@endcode
"""

if df is None:
    print("\nLoading preprocessed data...")
    df = pd.read_csv(OUTPUT_DIRS['data'] / 'all_beauty_preprocessed.csv')
    df['tokens'] = df['tokens'].apply(eval)

    print(f"Dataset size: {len(df)} reviews")

    if len(df) > sample_size:
        print(f"Sampling {sample_size} reviews for coherence testing...")
        df_sample = df.sample(n=sample_size, random_state=42)
    else:
        df_sample = df.copy()

```

```

print("\nCreating dictionary and corpus...")
dictionary = corpora.Dictionary(df_sample['tokens'])

print(f"Dictionary size before filtering: {len(dictionary)}")

dictionary.filter_extremes(
    no_below=10,
    no_above=0.5,
    keep_n=5000
)

print(f"Dictionary size after filtering: {len(dictionary)}")

corpus = [dictionary.doc2bow(tokens) for tokens in df_sample['tokens']]

dictionary.save(str(OUTPUT_DIRS['models'] / 'beauty_dictionary.dict'))
with open(OUTPUT_DIRS['models'] / 'beauty_corpus.pkl', 'wb') as f:
    pickle.dump(corpus, f)

print("\n" + "-"*70)
print("FINDING OPTIMAL NUMBER OF TOPICS")
print("-"*70)

model_list, coherence_values = compute_coherence_values(
    dictionary=dictionary,
    corpus=corpus,
    texts=df_sample['tokens'].tolist(),
    start=5,
    limit=16,
    step=1
)

num_topics = list(range(5, 16, 1))
plot_topic_coherence(num_topics, coherence_values)

optimal_idx = coherence_values.index(max(coherence_values))
optimal_topics = num_topics[optimal_idx]

print(f"\n Optimal number of topics: {optimal_topics}")
print(f" Coherence score: {coherence_values[optimal_idx]:.4f}")

results = {
    'num_topics': num_topics,
    'coherence_scores': coherence_values,
    'optimal_topics': optimal_topics
}

```

```

with open(OUTPUT_DIRS['models'] / 'coherence_results.pkl', 'wb') as f:
    pickle.dump(results, f)

return optimal_topics

def train_final_lda(df=None):
    """
    @brief Train final LDA model using optimal topic count

    @param df Optional DataFrame with preprocessed tokens. If None, loads from saved CSV

    @return gensim.models.LdaModel Trained LDA model
    """

    @details
    Steps performed:
    1. Loads preprocessed tokens and dictionary
    2. Retrieves optimal topic count from saved coherence results
    3. Trains final LDA model on the entire dataset
    4. Displays top words for each discovered topic
    5. Saves trained model and interactive visualization (pyLDAvis)

    @note
    Outputs saved to:
    - Model: `output/models/beauty_lda_final.model`
    - Visualization: `output/plots/lda/lda_visualization.html`

    @see find_optimal_topics(), LdaModel, gensimvis.prepare()

    @code{.py}
    lda_model = train_final_lda(df)
    lda_model.print_topics(5)
    @endcode
    """

    print("\n" + "-"*70)
    print("TRAINING FINAL LDA MODEL")
    print("-"*70)

    if df is None:
        print("\nLoading data...")
        df = pd.read_csv(OUTPUT_DIRS['data'] / 'all_beauty_preprocessed.csv')
        df['tokens'] = df['tokens'].apply(eval)

    dictionary = corpora.Dictionary.load(str(OUTPUT_DIRS['models'] / 'beauty_dictionary.dict'))

```

```

print("Creating corpus for full dataset...")
corpus = [dictionary.doc2bow(tokens) for tokens in df['tokens']]

with open(OUTPUT_DIRS['models'] / 'coherence_results.pkl', 'rb') as f:
    results = pickle.load(f)
    num_topics = results['optimal_topics']

print(f"\nTraining LDA model with {num_topics} topics...")

lda_model = LdaModel(
    corpus=corpus,
    id2word=dictionary,
    num_topics=num_topics,
    random_state=42,
    chunksize=2000,
    passes=15,
    iterations=400,
    alpha='auto',
    eta='auto',
    per_word_topics=True,
    eval_every=10
)

lda_model.save(str(OUTPUT_DIRS['models'] / 'beauty_lda_final.model'))

print("\n" + "-"*70)
print("DISCOVERED TOPICS")
print("-"*70)

for idx, topic in lda_model.print_topics(num_topics, num_words=15):
    print(f"\nTopic {idx}:")
    print(f"  {topic}")

topic_labels = {
    0: "General Feedback & Product Experience",
    1: "Wigs & Hairpieces",
    2: "Usage & Duration",
    3: "Skin Care & Sensitivity",
    4: "Product Reviews & Comparisons",
    5: "Positive Product Reviews",
    6: "Eye Makeup & Tools",
    7: "Product Value & Disappointment",
    8: "Cosmetics & Shades",
    9: "Fragrance & Scent",
    10: "Face Masks & Serums",
    11: "Oral Care & Toothbrushes",
}

```

```

12: "Nail Care & Polish",
13: "Hair Care & Styling",
14: "Product Fit & Size",
}

with open(OUTPUT_DIRS['models'] / 'topic_labels.pkl', 'wb') as f:
    pickle.dump(topic_labels, f)

print("\nCreating interactive visualization...")
vis = gensimvis.prepare(lda_model, corpus, dictionary, mds='mmds')
pyLDAvis.save_html(vis, str(OUTPUT_DIRS['plots'] / 'lda' / 
↳'lda_visualization.html'))

print(f"\n Model saved to {OUTPUT_DIRS['models'] / 'beauty_lda_final.
↳model'}")
print(f" Visualization saved to {OUTPUT_DIRS['plots'] / 'lda' / 
↳'lda_visualization.html'}")

return lda_model

def assign_topics(df=None):
    """
    @brief Assign dominant topic and probabilities to each review

    @param df Optional DataFrame with tokenized text. If None, loads from CSV

    @return pandas.DataFrame Reviews with topic assignments and probabilities

    @details
    For each review:
    - Loads final LDA model and dictionary
    - Computes topic distribution
    - Assigns dominant topic and maps it to a descriptive label
    - Saves merged dataset with topic probabilities

    @note
    Saves results to `output/data/all_beauty_with_topics.csv`

    @see get_document_topics(), LdaModel.load()

    @code{.py}
    df_with_topics = assign_topics(df)
    df_with_topics[['dominant_topic', 'topic_label']].head()
    @endcode
    """
    print("\n" + "-"*70)

```

```

print("ASSIGNING TOPICS TO REVIEWS")
print("-"*70)

if df is None:
    print("\nLoading data...")
    df = pd.read_csv(OUTPUT_DIRS['data'] / 'all_beauty_preprocessed.csv')
    df['tokens'] = df['tokens'].apply(eval)

dictionary = corpora.Dictionary.load(str(OUTPUT_DIRS['models'] /
                                         'beauty_dictionary.dict'))
lda_model = LdaModel.load(str(OUTPUT_DIRS['models'] / 'beauty_lda_final.
model'))

with open(OUTPUT_DIRS['models'] / 'topic_labels.pkl', 'rb') as f:
    topic_labels = pickle.load(f)

corpus = [dictionary.doc2bow(tokens) for tokens in df['tokens']]

print("\nProcessing documents...")
topics_df = get_document_topics(lda_model, corpus)

df = pd.concat([df.reset_index(drop=True), topics_df], axis=1)

df['topic_label'] = df['dominant_topic'].map(topic_labels)
df.loc[df['dominant_topic'] == -1, 'topic_label'] = 'Unknown'

df.to_csv(OUTPUT_DIRS['data'] / 'all_beauty_with_topics.csv', index=False)

print("\n" + "-"*70)
print("TOPIC ASSIGNMENT COMPLETE")
print("-"*70)
print("\nTopic distribution:")
print(df['topic_label'].value_counts())

print(f"\n Saved to {OUTPUT_DIRS['data'] / 'all_beauty_with_topics.csv'}")

return df


def naive_bayes_classification(df=None):
    """
    @brief Train and evaluate Naive Bayes sentiment classifier with SMOTE
    balancing

    @param df Optional DataFrame with topic probabilities. If None, loads from
    CSV
    """

```

```

@return pandas.DataFrame Test subset with predictions and true labels

@details
Complete supervised sentiment classification workflow:
1. Vectorizes text using TF-IDF (unigrams & bigrams)
2. Concatenates topic probability features
3. Balances training data using SMOTE
4. Trains multiple Naive Bayes variants and selects best based on F1-score
5. Performs hyperparameter tuning on the best model
6. Evaluates and compares baseline vs tuned performance
7. Generates confusion matrix and F1-score visualizations
8. Saves tuned model and vectorizer

@note
Saves:
- Model: `output/models/naive_bayes_model_balanced.pkl`
- Vectorizer: `output/models/tfidf_vectorizer.pkl`
- Predictions: `output/data/all_beauty_predictions_balanced.csv`

@see balance_dataset(), train_and_evaluate_models(), tune_hyperparameters(),
      plot_confusion_matrix_comparison(), plot_f1_comparison()

@code{.py}
df_test = naive_bayes_classification(df)
print(df_test[['sentiment', 'predicted_sentiment']].head())
@endcode
"""

if df is None:
    print("\nLoading data with topics...")
    df = pd.read_csv(OUTPUT_DIRS['data'] / 'all_beauty_with_topics.csv')

print(f"Dataset size: {len(df)}")
print("\nOriginal sentiment distribution:")
print(df['sentiment'].value_counts())

print("\nCreating TF-IDF features...")
tfidf_vectorizer = TfidfVectorizer(
    max_features=5000,
    min_df=5,
    max_df=0.7,
    ngram_range=(1, 2),
    sublinear_tf=True
)

```

```

X_tfidf = tfidf_vectorizer.fit_transform(df['processed_text'])

topic_prob_cols = [col for col in df.columns if col.startswith('topic_') and col.endswith('_prob')]
X_topics = df[topic_prob_cols].fillna(0).values

print("Combining features...")
X_combined = hstack([X_tfidf, X_topics])
print(f"Feature shape: {X_combined.shape}")

y = df['sentiment']

print("\nSplitting data...")
X_train, X_test, y_train, y_test, idx_train, idx_test = train_test_split(
    X_combined, y, df.index,
    test_size=0.2,
    random_state=42,
    stratify=y
)

print(f"Training set: {X_train.shape[0]} samples")
print(f"Test set: {X_test.shape[0]} samples")

print("\n" + "-"*70)
print("BALANCING DATASET WITH SMOTE")
print("-"*70)

X_train_balanced, y_train_balanced = balance_dataset(X_train, y_train)

print(f"\nBalanced training set: {len(y_train_balanced)} samples")
print(pd.Series(y_train_balanced).value_counts())

print("\n" + "-"*70)
print("TRAINING MODELS")
print("-"*70)

results = train_and_evaluate_models(X_train_balanced, y_train_balanced, X_test, y_test)

best_name = max(results, key=lambda k: results[k]['f1'])
best_model = results[best_name]['model']
BestModelClass = type(best_model)

print(f"\n Best model: {best_name} (F1: {results[best_name]['f1']:.4f})")

print("\n" + "-"*70)
print("HYPERPARAMETER TUNING")

```

```

print("-"*70)

tuned_model = tune_hyperparameters(BestModelClass, X_train_balanced, □
y_train_balanced)
y_pred_tuned = tuned_model.predict(X_test)

print("\n" + "-"*70)
print("EVALUATION")
print("-"*70)

accuracy_tuned = accuracy_score(y_test, y_pred_tuned)
f1_tuned = f1_score(y_test, y_pred_tuned, average='weighted')

print(f"\nOverall Performance:")
print(f" Accuracy: {accuracy_tuned:.4f}")
print(f" Weighted F1: {f1_tuned:.4f}")

print("\nDetailed Classification Report:")
print(classification_report(y_test, y_pred_tuned, digits=4))

baseline_model = BestModelClass(alpha=0.1)
baseline_model.fit(X_train, y_train)
y_pred_baseline = baseline_model.predict(X_test)

before_metrics = precision_recall_fscore_support(y_test, y_pred_baseline,
                                                 labels=['negative', □
'neutral', 'positive'])
after_metrics = precision_recall_fscore_support(y_test, y_pred_tuned,
                                                labels=['negative', □
'neutral', 'positive'])

print("\n" + "-"*70)
print("GENERATING VISUALIZATIONS")
print("-"*70)

plot_confusion_matrix_comparison(y_test, y_pred_baseline, y_pred_tuned)
plot_f1_comparison(before_metrics, after_metrics)

print("\n" + "-"*70)
print("SAVING MODELS")
print("-"*70)

joblib.dump(tuned_model, OUTPUT_DIRS['models'] / □
'naive_bayes_model_balanced.pkl')
joblib.dump(tfidf_vectorizer, OUTPUT_DIRS['models'] / 'tfidf_vectorizer.□
pkl')

```

```

    print(f" Model saved to {OUTPUT_DIRS['models']} /"
        ↪'naive_bayes_model_balanced.pkl'}")
    print(f" Vectorizer saved to {OUTPUT_DIRS['models']} / 'tfidf_vectorizer."
        ↪pkl'}")

    df_test = df.loc[idx_test].copy()
    df_test['predicted_sentiment'] = y_pred_tuned
    df_test.to_csv(OUTPUT_DIRS['data'] / 'all_beauty_predictions_balanced.csv', □
        ↪index=False)

    print(f" Predictions saved to {OUTPUT_DIRS['data']} /"
        ↪'all_beauty_predictions_balanced.csv')")

    save_smote_report(y_train, y_train_balanced, y_test, y_pred_baseline, □
        ↪y_pred_tuned)

    return df_test
}

def aspect_sentiment_analysis(df=None):
    """
    @brief Perform aspect-based sentiment analysis and visualization

    @param df Optional DataFrame with predicted sentiments. If None, loads from
    ↪CSV

    @return None

    @details
    Combines topic modeling and sentiment predictions to analyze sentiment per
    ↪aspect:
    - Computes sentiment distribution across discovered aspects
    - Identifies strongest and weakest aspects
    - Generates word clouds for top aspects
    - Analyzes sentiment trends over time
    - Saves aspect-wise sentiment summary report

    @note
    Saves report to `output/reports/aspect_summary.csv`

    @see plot_aspect_sentiment_distribution(), analyze_aspect(),
    ↪generate_aspect_wordclouds(),
    plot_sentiment_trends()

    @code{.py}

```

```

aspect_sentiment_analysis(df_test)
# Outputs visual reports and summary tables
@endcode
"""

if df is None:
    print("\nLoading predictions...")
    df = pd.read_csv(OUTPUT_DIRS['data'] / 'all_beauty_predictions_balanced.
    ↪csv')
print(f"Total reviews: {len(df)}")

df = df[df['topic_label'] != 'Unknown'].copy()

aspect_sentiment = pd.crosstab(
    df['topic_label'],
    df['predicted_sentiment'],
    normalize='index'
) * 100

print("\n" + "-"*70)
print("SENTIMENT DISTRIBUTION BY ASPECT")
print("-"*70)
print("\n", aspect_sentiment.round(2))

print("\nGenerating visualizations...")
plot_aspect_sentiment_distribution(df)

aspect_sentiment_counts = pd.crosstab(df['topic_label'], ↪
    ↪df['predicted_sentiment'])

aspect_summary = pd.DataFrame({
    'Total_Reviews': df.groupby('topic_label').size(),
    'Positive_%': aspect_sentiment['positive'],
    'Neutral_%': aspect_sentiment['neutral'],
    'Negative_%': aspect_sentiment['negative'],
    'Net_Sentiment': aspect_sentiment['positive'] - ↪
        ↪aspect_sentiment['negative']
})
aspect_summary = aspect_summary.sort_values('Net_Sentiment', ↪
    ↪ascending=False)

print("\n" + "-"*70)
print("ASPECT STRENGTHS & WEAKNESSES")
print("-"*70)

```

```

print("\nSTRONGEST ASPECTS:")
for aspect in aspect_summary.head(3).index:
    pos_pct = aspect_summary.loc[aspect, 'Positive_%']
    neg_pct = aspect_summary.loc[aspect, 'Negative_%']
    print(f" • {aspect}: {pos_pct:.1f}% positive, {neg_pct:.1f}% negative")

print("\nWEAKEST ASPECTS:")
for aspect in aspect_summary.tail(3).index:
    pos_pct = aspect_summary.loc[aspect, 'Positive_%']
    neg_pct = aspect_summary.loc[aspect, 'Negative_%']
    print(f" • {aspect}: {pos_pct:.1f}% positive, {neg_pct:.1f}% negative")

print("\n" + "-"*70)
print("DETAILED ASPECT ANALYSIS")
print("-"*70)

for aspect in aspect_summary.head(2).index:
    analyze_aspect(df, aspect)

print("\n" + "-"*70)
print("GENERATING WORD CLOUDS")
print("-"*70)

for aspect in aspect_summary.head(3).index:
    print(f" Generating word clouds for {aspect}...")
    generate_aspect_wordclouds(df, aspect)

print("\n" + "-"*70)
print("TEMPORAL ANALYSIS")
print("-"*70)

df['date'] = pd.to_datetime(df['timestamp'], unit='ms')
df['year_month'] = df['date'].dt.to_period('M')

print("\nPlotting overall sentiment trends...")
plot_sentiment_trends(df)

print("\nPlotting aspect-specific trends...")
for aspect in aspect_summary.head(2).index:
    print(f" {aspect}...")
    plot_sentiment_trends(df, aspect)

aspect_summary.to_csv(OUTPUT_DIRS['reports'] / 'aspect_summary.csv')
print(f"\n Summary saved to {OUTPUT_DIRS['reports'] / 'aspect_summary.
˓→csv'}")

```

```
print("\n" + "="*70)
print("ANALYSIS COMPLETE!")
print("="*70)
```

[6]: setup()

```
Downloading NLTK data...
Setup complete
```

[7]: df = load_and_explore('All_Beauty.jsonl.gz')

```
Loading reviews...
Progress: 700,000 reviews
Loaded 701,528 reviews
Saved to output/data/all_beauty_reviews.csv
```

```
Dataset shape: (701528, 10)
Columns: ['rating', 'title', 'text', 'images', 'asin', 'parent_asin', 'user_id',
'timestamp', 'helpful_vote', 'verified_purchase']
```

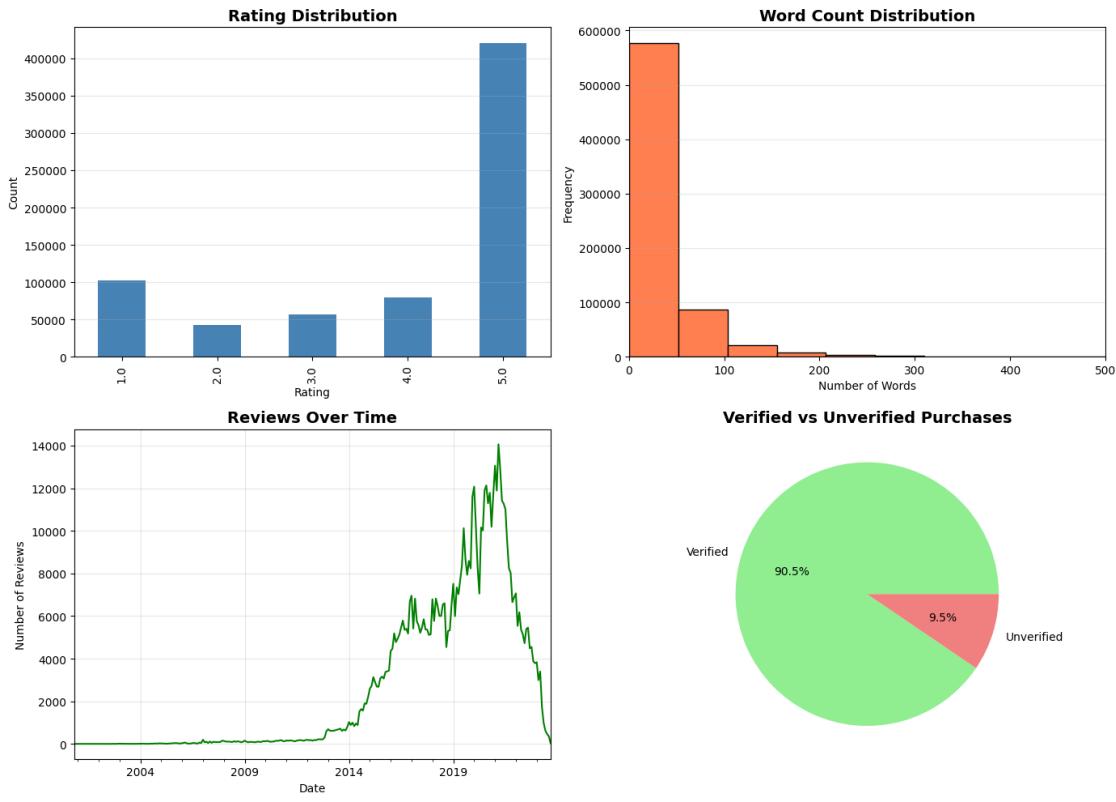
DATASET OVERVIEW

```
Total reviews: 701,528
Unique users: 631,986
Unique products: 112,565
Verified purchases: 634,969 (90.5%)
```

```
Rating distribution:
rating
1.0    102080
2.0    43034
3.0    56307
4.0    79381
5.0    420726
Name: count, dtype: int64
```

```
Average text length: 173 characters
Average word count: 33 words
```

```
Generating visualizations...
```



```
[8]: df = preprocess_data(df)
```

Original dataset: 701,528 reviews
After removing null text: 701,528 reviews

Cleaning text...
Tokenizing and lemmatizing...
After filtering short reviews: 530,546 reviews

PREPROCESSING COMPLETE

Final dataset: 530,546 reviews

Sentiment distribution:
sentiment
positive 367721
negative 116231
neutral 46594
Name: count, dtype: int64

Saved to output/data/all_beauty_preprocessed.csv

```
[9]: optimal_topics = find_optimal_topics(df)
```

Dataset size: 530,546 reviews

Sampling 100,000 reviews for coherence testing...

Creating dictionary and corpus...

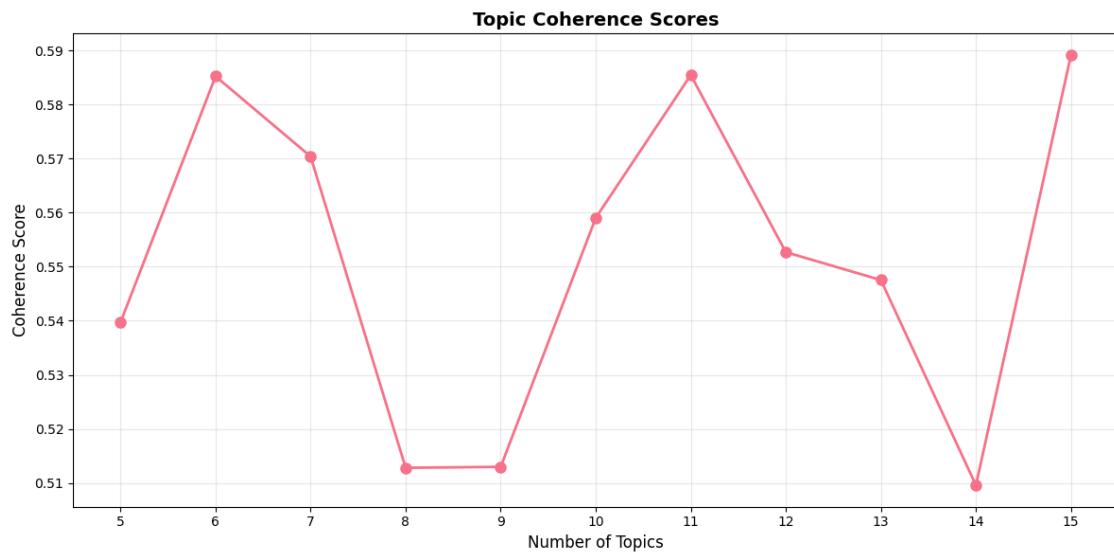
Dictionary size before filtering: 36,027

Dictionary size after filtering: 5,000

FINDING OPTIMAL NUMBER OF TOPICS

Testing 15 topics...

Tested 11 topic configurations



Optimal number of topics: 15

Coherence score: 0.5892

```
[10]: lda_model = train_final_lda(df)
```

TRAINING FINAL LDA MODEL

Creating corpus for full dataset...

Training LDA model with 15 topics...

DISCOVERED TOPICS

Topic 0:

0.111*"not" + 0.064*"like" + 0.038*"would" + 0.037*"get" + 0.023*"make" +
0.022*"even" + 0.022*"also" + 0.022*"much" + 0.021*"little" + 0.018*"feel" +
0.015*"better" + 0.014*"still" + 0.014*"way" + 0.013*"lot" + 0.012*"need"

Topic 1:

0.127*"look" + 0.065*"wig" + 0.042*"natural" + 0.036*"wear" + 0.034*"picture"
+ 0.022*"real" + 0.021*"looked" + 0.020*"black" + 0.017*"length" + 0.017*"style"
+ 0.016*"compliment" + 0.016*"full" + 0.015*"wearing" + 0.013*"inch" +
0.013*"dark"

Topic 2:

0.080*"use" + 0.064*"time" + 0.045*"day" + 0.040*"using" + 0.034*"first" +
0.031*"long" + 0.025*"put" + 0.024*"week" + 0.023*"last" + 0.021*"dry" +
0.019*"back" + 0.019*"take" + 0.018*"month" + 0.016*"every" + 0.016*"without"

Topic 3:

0.193*"skin" + 0.053*"oil" + 0.041*"smooth" + 0.029*"lotion" + 0.025*"feel" +
0.024*"sensitive" + 0.024*"feeling" + 0.024*"leaf" + 0.021*"body" +
0.017*"ingredient" + 0.017*"oily" + 0.016*"dry" + 0.015*"muy" + 0.015*"sticky" +
0.014*"greasy"

Topic 4:

0.114*"product" + 0.059*"one" + 0.036*"used" + 0.025*"got" + 0.022*"bought" +
0.017*"best" + 0.016*"year" + 0.016*"amazing" + 0.016*"tried" + 0.015*"received" +
0.014*"two" + 0.013*"give" + 0.013*"review" + 0.013*"never" +
0.013*"different"

Topic 5:

0.080*"love" + 0.072*"great" + 0.053*"good" + 0.048*"work" + 0.044*"really" +
0.038*"well" + 0.033*"easy" + 0.028*"quality" + 0.027*"buy" + 0.026*"recommend" +
0.026*"soft" + 0.026*"price" + 0.023*"definitely" + 0.023*"nice" +
0.020*"perfect"

Topic 6:

0.151*"lash" + 0.038*"eyelash" + 0.038*"shave" + 0.036*"razor" + 0.031*"eye" +
0.031*"lip" + 0.029*"eyeliner" + 0.029*"shaver" + 0.027*"blade" + 0.027*"powder" +
0.024*"mascara" + 0.021*"eyebrow" + 0.021*"liner" + 0.020*"shaving" +
0.017*"pair"

Topic 7:

0.067*"money" + 0.048*"worth" + 0.043*"disappointed" + 0.041*"item" +
0.030*"box" + 0.030*"waste" + 0.027*"package" + 0.027*"arrived" + 0.026*"cheap"

```
+ 0.024*"return" + 0.021*"broke" + 0.020*"glue" + 0.019*"broken" + 0.018*"half"  
+ 0.017*"horrible"
```

Topic 8:

```
0.225*"color" + 0.098*"bottle" + 0.078*"light" + 0.035*"spray" + 0.025*"white"  
+ 0.024*"red" + 0.024*"pink" + 0.021*"brown" + 0.020*"weight" + 0.017*"match" +  
0.017*"blend" + 0.017*"foundation" + 0.016*"blue" + 0.015*"pump" + 0.013*"shade"
```

Topic 9:

```
0.192*"smell" + 0.078*"scent" + 0.039*"soap" + 0.037*"strong" + 0.034*"beard"  
+ 0.022*"fragrance" + 0.018*"fresh" + 0.018*"deodorant" + 0.017*"perfume" +  
0.014*"bar" + 0.014*"christmas" + 0.014*"chemical" + 0.013*"birthday" +  
0.013*"body" + 0.013*"smelled"
```

Topic 10:

```
0.086*"face" + 0.043*"eye" + 0.039*"cream" + 0.038*"mask" + 0.027*"night" +  
0.027*"difference" + 0.021*"line" + 0.020*"noticed" + 0.018*"area" +  
0.017*"morning" + 0.015*"roller" + 0.015*"help" + 0.014*"serum" + 0.012*"spot" +  
0.012*"burn"
```

Topic 11:

```
0.188*"brush" + 0.068*"clean" + 0.030*"teeth" + 0.025*"battery" +  
0.024*"bristle" + 0.023*"handle" + 0.019*"toothbrush" + 0.018*"charge" +  
0.017*"cleaning" + 0.015*"power" + 0.014*"unit" + 0.012*"brushing" +  
0.011*"setting" + 0.011*"drill" + 0.010*"cleaner"
```

Topic 12:

```
0.223*"nail" + 0.087*"polish" + 0.085*"gel" + 0.041*"coat" + 0.030*"delivery"  
+ 0.025*"base" + 0.022*"professional" + 0.020*"top" + 0.018*"delivered" +  
0.017*"file" + 0.017*"beginner" + 0.016*"lamp" + 0.015*"glitter" + 0.015*"plate"  
+ 0.014*"sticker"
```

Topic 13:

```
0.433*"hair" + 0.037*"thick" + 0.032*"curl" + 0.025*"shampoo" + 0.021*"clip" +  
0.020*"comb" + 0.018*"shed" + 0.017*"conditioner" + 0.016*"heat" +  
0.015*"shedding" + 0.013*"straight" + 0.013*"tangle" + 0.013*"curly" +  
0.012*"iron" + 0.011*"thin"
```

Topic 14:

```
0.025*"head" + 0.025*"small" + 0.024*"keep" + 0.023*"size" + 0.023*"fit" +  
0.020*"enough" + 0.020*"hold" + 0.016*"easily" + 0.015*"big" + 0.015*"cut" +  
0.014*"bag" + 0.014*"plastic" + 0.013*"videoid" + 0.013*"part" + 0.012*"around"
```

Creating interactive visualization...

```
Model saved to output/models/beauty_lda_final.model  
Visualization saved to output/plots/lda/lda_visualization.html
```

```
[11]: df = assign_topics(df)
```

ASSIGNING TOPICS TO REVIEWS

Processing documents...
Processed 500,000 reviews
Topic assignment complete

TOPIC ASSIGNMENT COMPLETE

Topic distribution:

topic_label	count
General Feedback & Product Experience	218322
Positive Product Reviews	163061
Product Reviews & Comparisons	85482
Usage & Duration	26629
Product Fit & Size	18033
Hair Care & Styling	4636
Skin Care & Sensitivity	3520
Wigs & Hairpieces	2466
Face Masks & Serums	2285
Product Value & Disappointment	1542
Fragrance & Scent	1399
Cosmetics & Shades	1002
Oral Care & Toothbrushes	884
Nail Care & Polish	683
Eye Makeup & Tools	602

Name: count, dtype: int64

Saved to output/data/all_beauty_with_topics.csv

```
[12]: df_test = naive_bayes_classification(df)
```

Dataset size: 530,546

Original sentiment distribution:

sentiment	count
positive	367721
negative	116231
neutral	46594

Name: count, dtype: int64

Creating TF-IDF features...

```
Combining features...
Feature shape: (530546, 5015)
```

```
Splitting data...
Training set: 424,436 samples
Test set: 106,110 samples
```

```
BALANCING DATASET WITH SMOTE
```

```
Applying SMOTE...
```

```
/home/varunadhityagb/Syncthing/absa-on-amazon-reviews/.venv/lib/python3.9/site-
packages/sklearn/base.py:474: FutureWarning: `BaseEstimator._validate_data` is
deprecated in 1.6 and will be removed in 1.7. Use
`sklearn.utils.validation.validate_data` instead. This function becomes public
and is part of the scikit-learn developer API.
    warnings.warn(
/home/varunadhityagb/Syncthing/absa-on-amazon-reviews/.venv/lib/python3.9/site-
packages/sklearn/utils/_tags.py:354: DeprecationWarning: The SMOTE or classes
from which it inherits use `'_get_tags` and `'_more_tags` . Please define the
`'_sklearn_tags__` method, or inherit from `sklearn.base.BaseEstimator` and/or
other appropriate mixins such as `sklearn.base.TransformerMixin` ,
`sklearn.base.ClassifierMixin` , `sklearn.base.RegressorMixin` , and
`sklearn.base.OutlierMixin` . From scikit-learn 1.7, not defining
`'_sklearn_tags__` will raise an error.
    warnings.warn(
```

```
Applying undersampling...
```

```
/home/varunadhityagb/Syncthing/absa-on-amazon-reviews/.venv/lib/python3.9/site-
packages/sklearn/utils/deprecation.py:151: FutureWarning: 'force_all_finite' was
renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.
    warnings.warn(
/home/varunadhityagb/Syncthing/absa-on-amazon-reviews/.venv/lib/python3.9/site-
packages/sklearn/base.py:484: FutureWarning: `BaseEstimator._check_n_features` 
is deprecated in 1.6 and will be removed in 1.7. Use
`sklearn.utils.validation._check_n_features` instead.
    warnings.warn(
/home/varunadhityagb/Syncthing/absa-on-amazon-reviews/.venv/lib/python3.9/site-
packages/sklearn/base.py:493: FutureWarning:
`BaseEstimator._check_feature_names` is deprecated in 1.6 and will be removed in
1.7. Use `sklearn.utils.validation._check_feature_names` instead.
    warnings.warn(
```

```
Balanced training set: 330,000 samples
```

```
sentiment
positive    150000
negative    100000
neutral     80000
Name: count, dtype: int64
```

TRAINING MODELS

```
Training MultinomialNB... (F1: 0.8144)
Training ComplementNB... (F1: 0.8046)
```

```
Best model: MultinomialNB (F1: 0.8144)
```

HYPERPARAMETER TUNING

```
Performing grid search... (Best F1: 0.7279)
```

EVALUATION

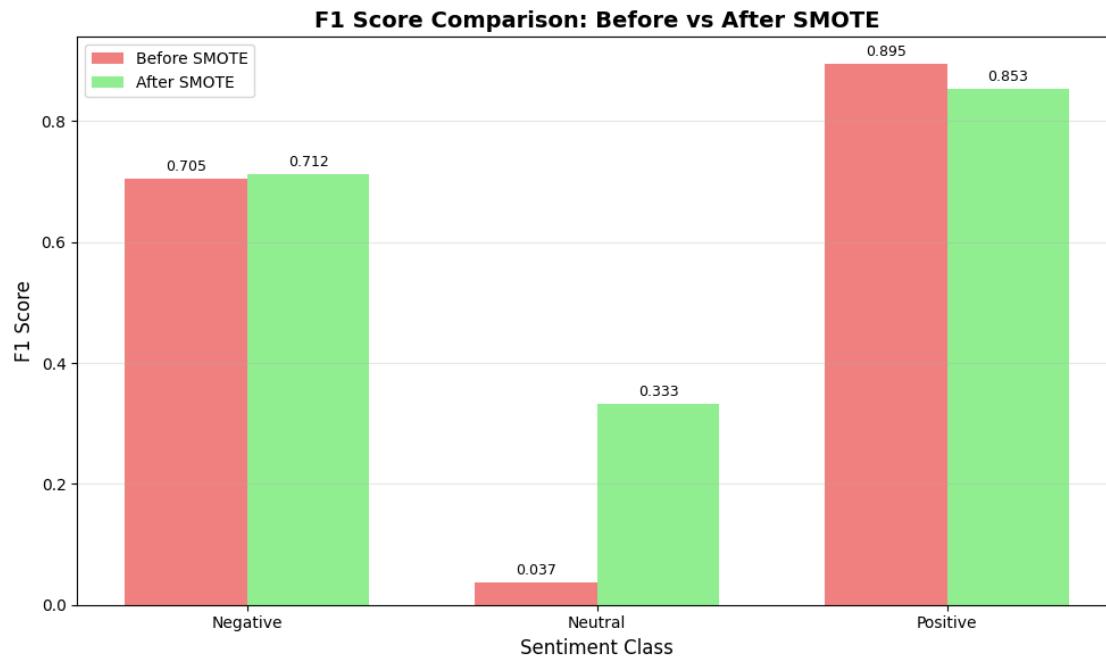
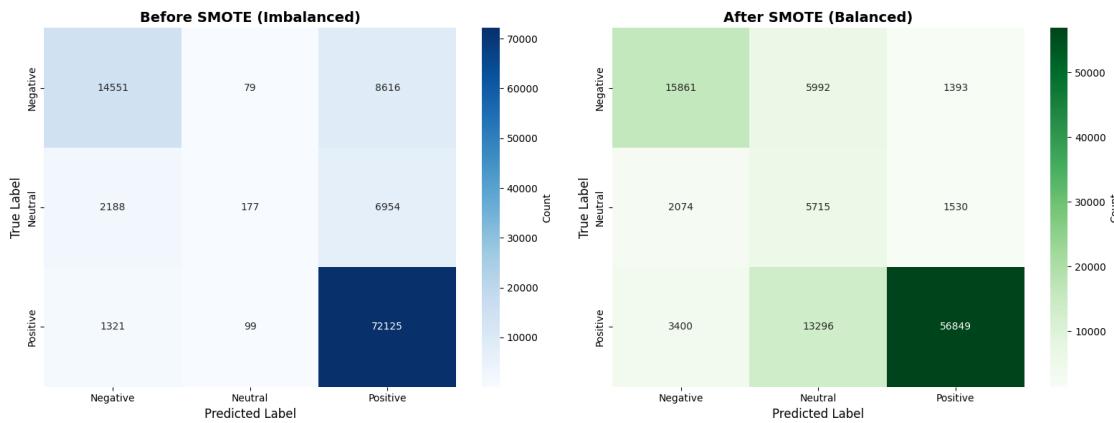
Overall Performance:

```
Accuracy: 0.7391
Weighted F1: 0.7762
```

Detailed Classification Report:

	precision	recall	f1-score	support
negative	0.7434	0.6823	0.7116	23246
neutral	0.2286	0.6133	0.3330	9319
positive	0.9511	0.7730	0.8528	73545
accuracy			0.7391	106110
macro avg	0.6410	0.6895	0.6325	106110
weighted avg	0.8421	0.7391	0.7762	106110

GENERATING VISUALIZATIONS



SAVING MODELS

Model saved to output/models/naive_bayes_model_balanced.pkl
 Vectorizer saved to output/models/tfidf_vectorizer.pkl
 Predictions saved to output/data/all_beauty_predictions_balanced.csv
 Report saved: output/reports/smote_improvement_report.txt

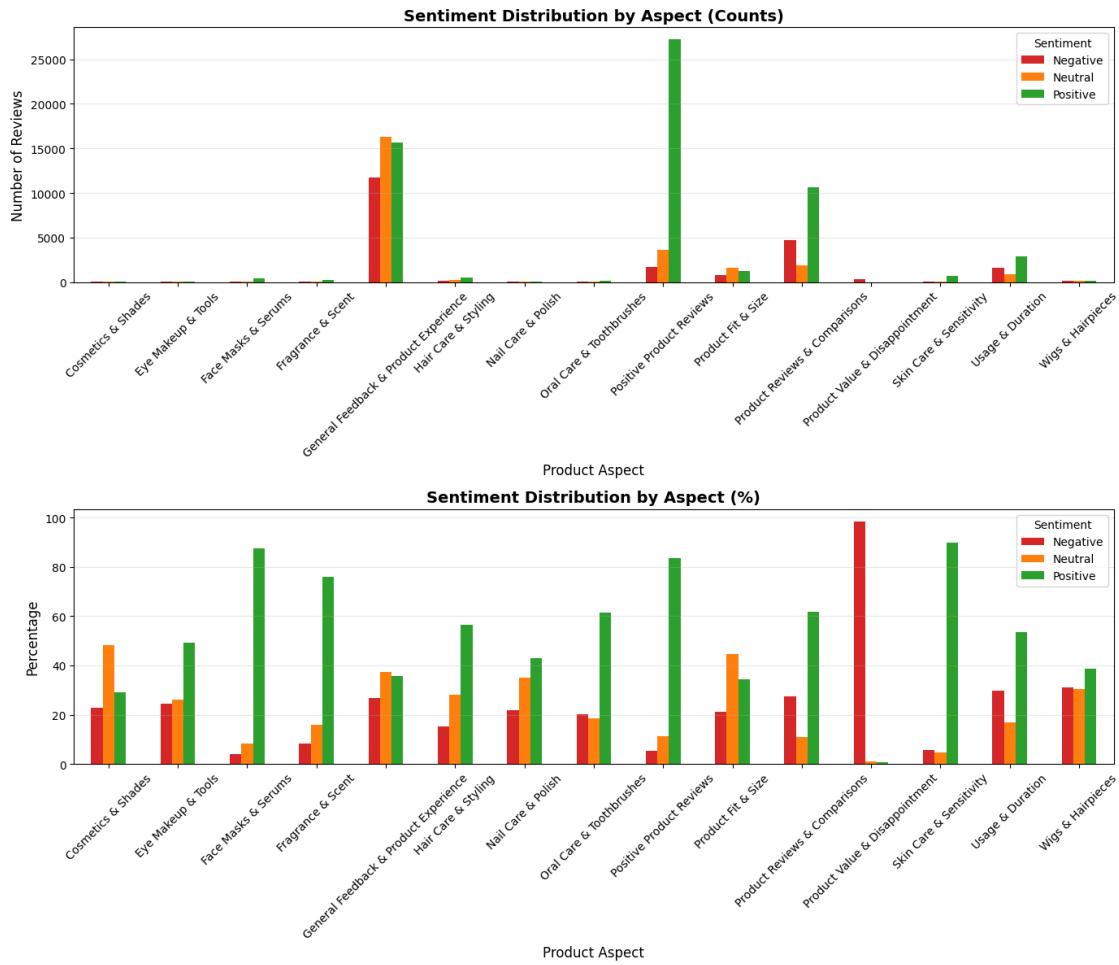
[13]: aspect_sentiment_analysis(df_test)

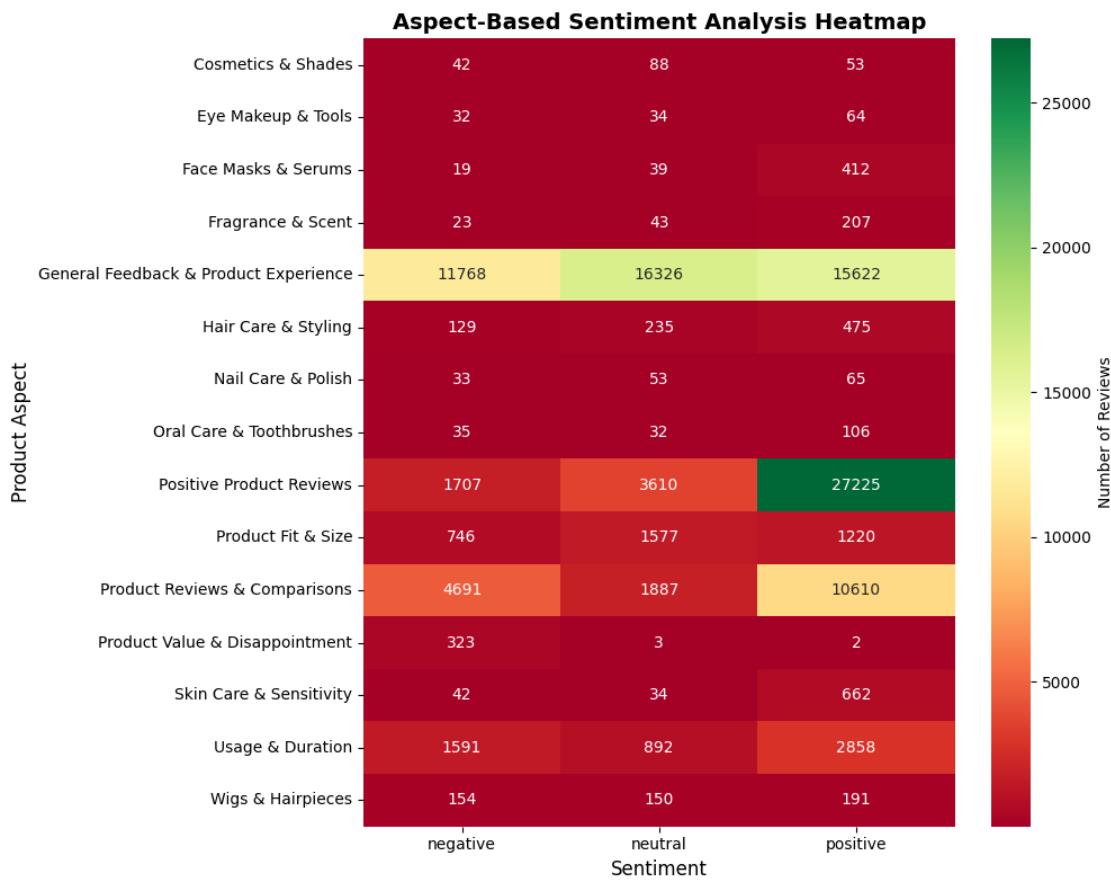
Total reviews: 106,110

SENTIMENT DISTRIBUTION BY ASPECT

predicted_sentiment topic_label	negative	neutral	positive
Cosmetics & Shades	22.95	48.09	28.96
Eye Makeup & Tools	24.62	26.15	49.23
Face Masks & Serums	4.04	8.30	87.66
Fragrance & Scent	8.42	15.75	75.82
General Feedback & Product Experience	26.92	37.35	35.74
Hair Care & Styling	15.38	28.01	56.62
Nail Care & Polish	21.85	35.10	43.05
Oral Care & Toothbrushes	20.23	18.50	61.27
Positive Product Reviews	5.25	11.09	83.66
Product Fit & Size	21.06	44.51	34.43
Product Reviews & Comparisons	27.29	10.98	61.73
Product Value & Disappointment	98.48	0.91	0.61
Skin Care & Sensitivity	5.69	4.61	89.70
Usage & Duration	29.79	16.70	53.51
Wigs & Hairpieces	31.11	30.30	38.59

Generating visualizations...





ASPECT STRENGTHS & WEAKNESSES

STRONGEST ASPECTS:

- Skin Care & Sensitivity: 89.7% positive, 5.7% negative
- Face Masks & Serums: 87.7% positive, 4.0% negative
- Positive Product Reviews: 83.7% positive, 5.2% negative

WEAKEST ASPECTS:

- Wigs & Hairpieces: 38.6% positive, 31.1% negative
- Cosmetics & Shades: 29.0% positive, 23.0% negative
- Product Value & Disappointment: 0.6% positive, 98.5% negative

DETAILED ASPECT ANALYSIS

ASPECT: Skin Care & Sensitivity

Total reviews: 738

Sentiment Distribution:

Positive :	662 (89.7%)
Neutral :	34 (4.6%)
Negative :	42 (5.7%)

Average Rating: 4.57/5.0

POSITIVE EXAMPLES:

[1] Rating: 5/5 | Confidence: 0.17

"I have been using this facial cleanser with the toner and moisturizer for a few weeks now and I have to say that there is a noticeable difference with my skin. My pores are smaller, my skin is lovely..."

[2] Rating: 5/5 | Confidence: 0.23

"I have been purchasing Shea Butter for over 2 decades for manufacturing body products and soap...I ran out unexpectedly and needed some Shea Butter quickly...I took a chance with this and it paid off!..."

NEGATIVE EXAMPLES:

[1] Rating: 3/5 | Confidence: 0.21

"The seal was broken under the lid when I received the package causing all of the coconut oil liquid to leak out leaving me with a dry scrub. Beware this is a scrub containing salt and you can certainl..."

[2] Rating: 5/5 | Confidence: 0.15

"Me gustaron los acabados y detalles, pero la senti muy apretada, deberia tener algo para ajustarla al tamaño de la cabeza."

NEUTRAL EXAMPLES:

[1] Rating: 5/5 | Confidence: 0.19

"Nice quality 16 oz bottle of coconut carrier oil.
Very clear with no discoloration or impurities visible.
Has a little 'soap' like smell I have never noticed with coconut oils. But b..."

[2] Rating: 2/5 | Confidence: 0.23

"I have dry, flakey skin area, read the reviews, and thought I would give it a shot. It didn't work for me. The product goes on smooth but didn't clear up the flaky skin. No oily reside left behind"

ASPECT: Face Masks & Serums

Total reviews: 470

Sentiment Distribution:

Positive :	412 (87.7%)
Neutral :	39 (8.3%)
Negative :	19 (4.0%)

Average Rating: 4.56/5.0

POSITIVE EXAMPLES:

[1] Rating: 5/5 | Confidence: 0.25

"I suffer from Blepharitis & eczema of the eyes and rosacea on my face and all the Facenearth products have helped more than any of the hundreds of dollars I have spent on other over the counter an..."

[2] Rating: 5/5 | Confidence: 0.19

"I love this eye cream. It feels gentle around my eyes. It has been working well for making my dark circles lighter. I fit this into my nightly skin care routine now."

NEGATIVE EXAMPLES:

[1] Rating: 2/5 | Confidence: 0.23

"I developed a rash after using this for about 2weeks so I stopped using it. In that time I did not notice an improvement in the spots on my face."

[2] Rating: 1/5 | Confidence: 0.19

"I am really disappointed. The reviews looked great, but these products caused so much irritation and burning on my face. I tried using them once at night and once in the morning. Definitely will be se..."

NEUTRAL EXAMPLES:

[1] Rating: 5/5 | Confidence: 0.21

"It actually works, per my wife. The eye mask is thicker than other brand she uses. After leave mask under eye for a couple of hours, since there's no instructions say how long it supposed to stay on..."

[2] Rating: 5/5 | Confidence: 0.22

"I've been using Flawless for about a month and I do see a slight improvement. Hopefully this will continue until the scar is completely gone."

GENERATING WORD CLOUDS

Generating word clouds for Skin Care & Sensitivity...

Word Clouds: Skin Care & Sensitivity



Generating word clouds for Face Masks & Serums...

Word Clouds: Face Masks & Serums



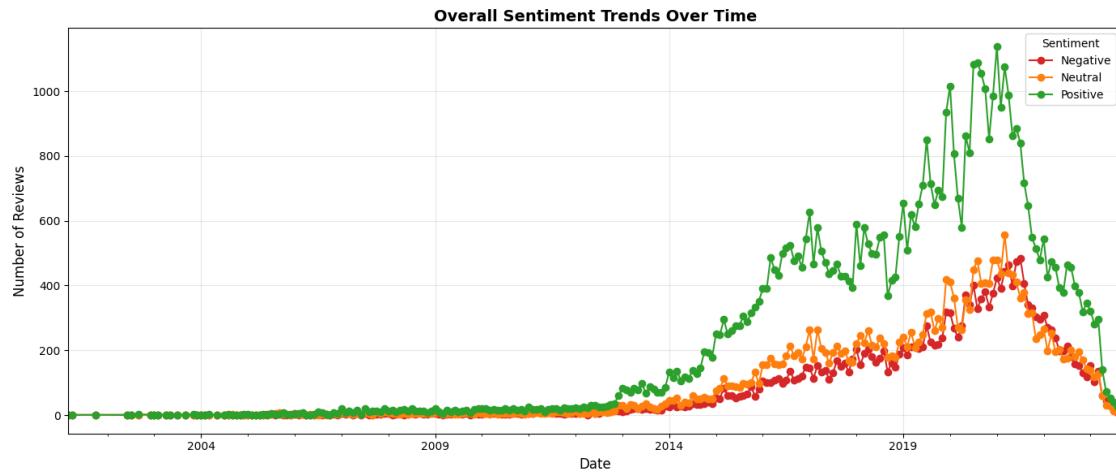
Generating word clouds for Positive Product Reviews...

Word Clouds: Positive Product Reviews



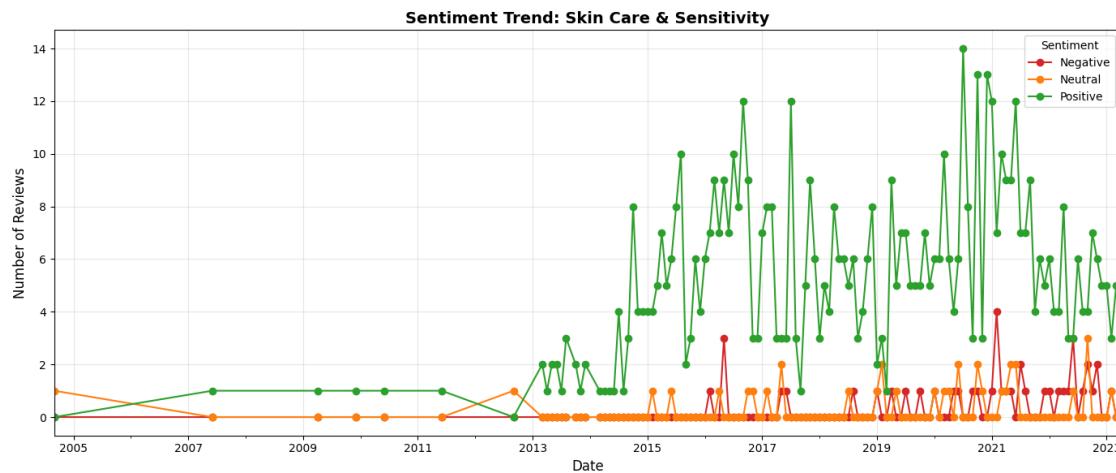
TEMPORAL ANALYSIS

Plotting overall sentiment trends...

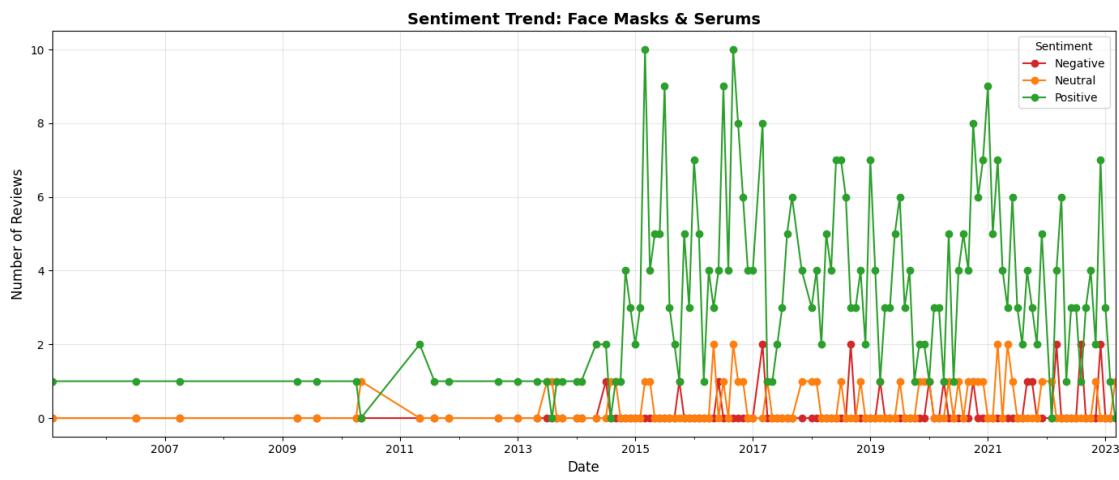


Plotting aspect-specific trends...

Skin Care & Sensitivity...



Face Masks & Serums...



Summary saved to output/reports/aspect_summary.csv

=====
ANALYSIS COMPLETE!
=====