

NLP SENTIMENT CLASSIFICATION PROJECT

Apple & Google Product Sentiment

Authors: Group 7 Members:

Brian Kiprop

Michael Mumina

Jeremiah Bii

Dennis Irimu

Emmanuel Kipleting

Date: 07/12/2025

1. Business Understanding

Organizations such as **Apple, Google, and third-party consumer electronics companies** rely heavily on public opinion expressed on social media platforms like Twitter (X). These organizations face the ongoing challenge of understanding how customers feel about their products in real time. With millions of tweets created every day, it becomes difficult for product teams, customer experience teams, and marketing departments to manually monitor sentiment or identify trends quickly.

Stakeholder Analysis

- **Primary Stakeholder:** Product and Marketing Teams at Apple and Google.
- **Secondary Stakeholders:**
 - Customer experience analysts
 - Social media managers
 - Competitor intelligence teams
 - Consumer behavior researchers

Problem Statement

Stakeholders need to answer crucial questions such as:

- *“How are users reacting to our new feature or product launch?”*
- *“Are negative emotions suddenly increasing around a particular product?”*
- *“Which categories attract the most positive engagement?”*

Manually reviewing thousands of tweets is impractical. Without automation, stakeholders risk:

- Missing early warning signs of dissatisfaction
- Failing to capitalize on positive sentiment

- Responding slowly to product or service issues raised by customers

Proposed Solution

This project develops an **NLP(Natural Language Processing) sentiment classification model** that automatically identifies whether a tweet expresses positive, negative, or neutral sentiment toward Apple or Google products.

This model provides stakeholders with:

- A scalable, automated way to track brand sentiment
- Faster insights for decision-making
- Clear patterns in consumer reactions to product updates, launches, and issues

Value Explanation

The value of the project lies in enabling stakeholders to:

- Detect sentiment shifts early
- Prioritize product improvements based on user feedback
- Tailor marketing strategies to public perception
- Benchmark Apple versus Google in terms of public sentiment

This reflects a genuine real-world need: technology companies actively use automated sentiment monitoring to understand and respond to the voice of the customer.

Future Scope

The workflow developed here lays the foundation for a more socially impactful project: analyzing tweets for **signs of depression or emotional distress**. Both projects require:

- Robust text preprocessing
- Feature engineering
- Classification modeling
- Interpretability and responsible AI principles

By addressing a real-world industry problem now, this project builds the technical and analytical skills required for a health-oriented NLP system in the future.

2. Environment Setup

```
In [1]: # Data Handling  
import pandas as pd  
import numpy as np  
  
# Visualization  
import matplotlib.pyplot as plt  
import seaborn as sns  
  
# Text Processing  
import re  
import nltk  
from nltk.corpus import stopwords  
from nltk.stem import WordNetLemmatizer  
  
# Modelling  
from sklearn.model_selection import train_test_split  
from sklearn.preprocessing import LabelEncoder, StandardScaler  
from sklearn.feature_extraction.text import TfidfVectorizer  
from sklearn.linear_model import LogisticRegression  
from sklearn.naive_bayes import MultinomialNB  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.svm import SVC  
from sklearn.svm import LinearSVC  
from imblearn.pipeline import Pipeline  
from sklearn.model_selection import GridSearchCV  
from imblearn.over_sampling import SMOTE  
  
# Evaluation  
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1  
from sklearn.metrics import confusion_matrix, classification_report  
from sklearn.metrics import precision_recall_fscore_support  
  
# Deep Learning  
import tensorflow as tf  
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Embedding, LSTM, Bidirectional, Dense, Dr  
from tensorflow.keras.preprocessing.text import Tokenizer  
from tensorflow.keras.preprocessing.sequence import pad_sequences  
  
from sklearn.feature_extraction.text import CountVectorizer  
  
from wordcloud import WordCloud  
  
# General  
import warnings  
warnings.filterwarnings("ignore")  
  
nltk.download("stopwords")  
nltk.download("wordnet")
```

```
[nltk_data] Downloading package stopwords to  
[nltk_data] C:\Users\Admin\AppData\Roaming\nltk_data...  
[nltk_data] Package stopwords is already up-to-date!  
[nltk_data] Downloading package wordnet to  
[nltk_data] C:\Users\Admin\AppData\Roaming\nltk_data...  
[nltk_data] Package wordnet is already up-to-date!
```

Out[1]: True

3. Data Understanding

3.1: Data Overview

The dataset used in this analysis consists of **9,093 tweets** collected for the purpose of analyzing sentiment toward technology brands. It contains three main features:

- **tweet_text** – the full content of the tweet.
- **emotion_in_tweet_is_directed_at** – the brand or product referenced in the tweet (e.g., Apple, Google).
- **is_there_an_emotion_directed_at_a_brand_or_product** – indicates whether the tweet expresses an emotion toward a brand/product.

The dataset is structured for a **multi-class classification problem**, where the target variable is the type of emotion expressed in the tweet. It combines textual content with brand related metadata, enabling the development of NLP models to detect sentiment and emotion.

The data is publicly available and imported using pandas

3.2: Data Description

3.2.1: Importing the dataset

```
In [2]: data = pd.read_csv('https://query.data.world/s/3r3b3chhfpyo7545c4regquyxcmc34  
                        encoding='latin1')
```

In [3]: `data.head()`

Out[3]:

	tweet_text	emotion_in_tweet_is_directed_at	is_there_an_emotion_directed_at_a_brand_or_pro
0	.@wesley83 I have a 3G iPhone. After 3 hrs twe...	iPhone	Negative en
1	@jessedee Know about @fludapp ? Awesome iPad/i...	iPad or iPhone App	Positive en
2	@swonderlin Can not wait for #iPad 2 also. The...	iPad	Positive en
3	@sxsw I hope this year's festival isn't as cra...	iPad or iPhone App	Negative en
4	@sxtxstate great stuff on Fri #SXSW: Marissa M...	Google	Positive en

3.2.2: Basic Structure

In [4]: `#data shape`
`data.shape`

Out[4]: (9093, 3)

In [5]: `#data columns`
`data.columns`

Out[5]: Index(['tweet_text', 'emotion_in_tweet_is_directed_at',
'is_there_an_emotion_directed_at_a_brand_or_product'],
dtype='object')

3.2.3: Overview of column types and non-null values

In [6]: `data.info()`

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9093 entries, 0 to 9092
Data columns (total 3 columns):
 #   Column                                                                 Non-Null Count  Dtype
---  ---
 0   tweet_text                                                            9092 non-null   object
 1   emotion_in_tweet_is_directed_at                                       3291 non-null   object
 2   is_there_an_emotion_directed_at_a_brand_or_product                 9093 non-null   object
dtypes: object(3)
memory usage: 213.2+ KB
```

3.2.4: Summary statistics Categorical

In [7]: `data.describe(include='O')`

Out[7]:

	tweet_text	emotion_in_tweet_is_directed_at	is_there_an_emotion_directed_at_a_brand_or
count	9092	3291	
unique	9065	9	
top	RT @mention Marissa Mayer: Google Will Connect...	iPad	No emotion toward brand c
freq	5	946	

3.2.5: Missing values

In [8]: `data.isna().mean()*100`

```
Out[8]: tweet_text                0.010997
emotion_in_tweet_is_directed_at  63.807324
is_there_an_emotion_directed_at_a_brand_or_product  0.000000
dtype: float64
```

3.2.6: Duplicates

```
In [9]: data.duplicated().sum()
```

```
Out[9]: 22
```

3.3: Data Summary

The dataset consists of **9,093 tweets** and includes three main features:

- **tweet_text** – the full tweet content
- **emotion_in_tweet_is_directed_at** – the brand or product referenced
- **is_there_an_emotion_directed_at_a_brand_or_product** – whether the tweet expresses emotion toward a brand/product

The data basic structure and quality:

- The **tweet_text** column is almost complete, with **9,092 non-null entries** and **9,065 unique tweets**, indicating very high diversity and minimal missing data.
- The **emotion_in_tweet_is_directed_at** column shows significant sparsity, with **63.8% missing values**, suggesting that many tweets do not mention a specific brand. Among the populated entries, there are **9 unique categories**, with **iPad** being the most frequently referenced.
- The **emotion-indicator** column is fully complete (**0% missing**) with **4 distinct categories**, where the most common label is **“No emotion toward brand or product”**.
- Only **22 duplicate records** are present, which is less than **0.25%** of the dataset, making it largely clean and ready for further preprocessing.

Overall This dataset provides a solid foundation for objectives outlined i.e. Emotion classification, Brand sentiment analysis, NLP-based tweet modeling. Despite the sparsity in brand-target information, the dataset contains **rich textual content** and **reliable emotion labels** that support exploratory analysis and machine learning applications.

4: Data Preparation

4.1: Data Cleaning

4.1.1: Making a copy of the data

```
In [10]: #copy
df = data.copy()
df.head()
```

Out[10]:

	tweet_text	emotion_in_tweet_is_directed_at	is_there_an_emotion_directed_at_a_brand_or_pro
0	.@wesley83 I have a 3G iPhone. After 3 hrs twe...	iPhone	Negative en
1	@jessedee Know about @fludapp ? Awesome iPad/i...	iPad or iPhone App	Positive en
2	@swonderlin Can not wait for #iPad 2 also. The...	iPad	Positive en
3	@sxsw I hope this year's festival isn't as cra...	iPad or iPhone App	Negative en
4	@sxtxstate great stuff on Fri #SXSW: Marissa M...	Google	Positive en

4.1.2: Removing Duplicates

```
In [11]: # Checking the number of rows before deduplication
print(f"Shape Before: {df.shape}")

#rmv duplicates
df = df.drop_duplicates()

#resetting index
df = df.reset_index(drop=True)

# Checking the number of rows after deduplication
print(f"Shape After: {df.shape}")
```

Shape Before: (9093, 3)
Shape After: (9071, 3)

- Removed: **22 duplicate rows**
- Status: **Dataset is ready for further preprocessing**

4.1.3: Handling missing values and standardizing columns

```
In [12]: #checking 'emotion_in_tweet_is_directed_at'
df['emotion_in_tweet_is_directed_at'].value_counts()
```

```
Out[12]: emotion_in_tweet_is_directed_at
iPad                                945
Apple                              659
iPad or iPhone App                 469
Google                             428
iPhone                             296
Other Google product or service    293
Android App                         80
Android                             77
Other Apple product or service     35
Name: count, dtype: int64
```

```
In [13]: #renaming columns
df = df.rename(columns={
    'tweet_text': 'tweet',
    'emotion_in_tweet_is_directed_at': 'target_product',
    'is_there_an_emotion_directed_at_a_brand_or_product': 'sentiment_label'
})

df.head()
```

```
Out[13]:
```

	tweet	target_product	sentiment_label
0	.@wesley83 I have a 3G iPhone. After 3 hrs twe...	iPhone	Negative emotion
1	@jessedee Know about @fludapp ? Awesome iPad/i...	iPad or iPhone App	Positive emotion
2	@swonderlin Can not wait for #iPad 2 also. The...	iPad	Positive emotion
3	@sxsxw I hope this year's festival isn't as cra...	iPad or iPhone App	Negative emotion
4	@sxtxstate great stuff on Fri #SXSW: Marissa M...	Google	Positive emotion

```
In [14]: #removing the null values on all columns for simplicity
df = df.dropna().reset_index(drop=True)

#cleanup validation
df.isna().sum()
```

```
Out[14]: tweet            0
target_product          0
sentiment_label         0
dtype: int64
```

```
In [15]: df['target_product'].unique()
```

```
Out[15]: array(['iPhone', 'iPad or iPhone App', 'iPad', 'Google', 'Android',
                'Apple', 'Android App', 'Other Google product or service',
                'Other Apple product or service'], dtype=object)
```

```

In [16]: ➤ apple_terms = ['apple', 'iphone', 'ipad', 'mac', 'ios']
google_terms = ['google', 'android', 'nexus', 'pixel']

# Convert the product label to lowercase
df['target_product'] = df['target_product'].str.lower()

# Helper function to classify each row
def classify_brand(text):
    if any(term in text for term in apple_terms):
        return 'apple'
    if any(term in text for term in google_terms):
        return 'google'
    return 'other' # In case something unexpected slips through

# Apply classification
df['company'] = df['target_product'].apply(classify_brand)

df.head()

```

Out[16]:

	tweet	target_product	sentiment_label	company
0	.@wesley83 I have a 3G iPhone. After 3 hrs twe...	iphone	Negative emotion	apple
1	@jessedee Know about @fludapp ? Awesome iPad/i...	ipad or iphone app	Positive emotion	apple
2	@swonderlin Can not wait for #iPad 2 also. The...	ipad	Positive emotion	apple
3	@sxsxw I hope this year's festival isn't as cra...	ipad or iphone app	Negative emotion	apple
4	@sxtxstate great stuff on Fri #SXSW: Marissa M...	google	Positive emotion	google

```

In [17]: ➤ df['company'].unique()

```

Out[17]: array(['apple', 'google'], dtype=object)

```

In [18]: ➤ df['sentiment_label'].unique()

```

Out[18]: array(['Negative emotion', 'Positive emotion',
'No emotion toward brand or product', 'I can't tell'], dtype=object)

```

In [19]: ➤ df['sentiment_label'] = df['sentiment_label'].replace({
    'No emotion toward brand or product': 'Neutral'
})

```

Handling Missing Values & Standardizing Columns Summary

- Reviewed the distribution of product labels to understand category frequency.
- Standardized column names for clarity (tweet , target_product , sentiment_label).
- Removed all rows with missing values to ensure a complete dataset.
- Converted product labels to lowercase for consistency.
- Mapped each product reference to its parent brand (Apple or Google), creating a new company column.

- Renamed the long sentiment label *"No emotion toward brand or product"* to the standardized category **"Neutral"** for clarity and alignment with sentiment analysis conventions.
- Final dataset contains only two company categories: **apple** and **google**.

4.1.4: Normalizing text (lowercase, remove URLs, mentions, hashtags)

```
In [20]: ▶ def clean_text(text):
            text = str(text).lower()                                # Lowercase

            # Correct URL removal
            text = re.sub(r'http\S+|www\S+', '', text)              # Remove URLs

            text = re.sub(r'@\w+', '', text)                        # Remove mentions
            text = re.sub(r'#', '', text)                          # Remove #
            text = re.sub(r'^a-z\s', ' ', text)                    # Remove punctuation
            text = re.sub(r'\s+', ' ', text).strip()               # Remove extra whitespace
            return text
```

4.1.5: Removing stopwords

```
In [21]: ▶ stop_words = set(stopwords.words('english'))
def preprocess_text(text):
    text = clean_text(text) # your cleaning function
    words = text.split()
    words = [w for w in words if w not in stop_words]
    return " ".join(words)
df['processed_text'] = df['tweet'].apply(preprocess_text)
```

```
In [22]: ▶ df.head()
```

Out[22]:

	tweet	target_product	sentiment_label	company	processed_text
0	.@wesley83 I have a 3G iPhone. After 3 hrs twe...	iphone	Negative emotion	apple	g iphone hrs tweeting rise austin dead need up...
1	@jessedee Know about @fludapp ? Awesome iPad/i...	ipad or iphone app	Positive emotion	apple	know awesome ipad iphone app likely appreciate...
2	@swonderlin Can not wait for #iPad 2 also. The...	ipad	Positive emotion	apple	wait ipad also sale sxsw
3	@sxsw I hope this year's festival isn't as cra...	ipad or iphone app	Negative emotion	apple	hope year festival crashy year iphone app sxsw
4	@sxtxstate great stuff on Fri #SXSW: Marissa M...	google	Positive emotion	google	great stuff fri sxsw marissa mayer google tim ...

4.1.6: Lemmatize words

```
In [23]: ▶ lemmatizer = WordNetLemmatizer()
def lemmatize_text(text):
    words = text.split()
    lem_words = [lemmatizer.lemmatize(w) for w in words]
    return " ".join(lem_words)
df['processed_text'] = df['processed_text'].apply(lemmatize_text)
```

```
In [24]: ▶ df.head()
```

Out[24]:

	tweet	target_product	sentiment_label	company	processed_text
0	.@wesley83 I have a 3G iPhone. After 3 hrs twe...	iphone	Negative emotion	apple	g iphone hr tweeting rise austin dead need upg...
1	@jessedee Know about @fludapp ? Awesome iPad/i...	ipad or iphone app	Positive emotion	apple	know awesome ipad iphone app likely appreciate...
2	@swonderlin Can not wait for #iPad 2 also. The...	ipad	Positive emotion	apple	wait ipad also sale sxsw
3	@sxsw I hope this year's festival isn't as cra...	ipad or iphone app	Negative emotion	apple	hope year festival crashy year iphone app sxsw
4	@sxtxstate great stuff on Fri #SXSW: Marissa M...	google	Positive emotion	google	great stuff fri sxsw marissa mayer google tim ...

Text Normalization & Preprocessing Summary

- Normalized all tweets by converting text to lowercase and removing URLs, mentions, hashtags, punctuation, and extra whitespace.
- Removed English stopwords to keep only meaningful words that contribute to sentiment.
- Applied lemmatization to reduce words to their base form, improving consistency across the dataset.
- Created a new column, `processed_text`, containing fully cleaned, stopword-free, and lemmatized tweet content prepared for feature extraction and modeling.

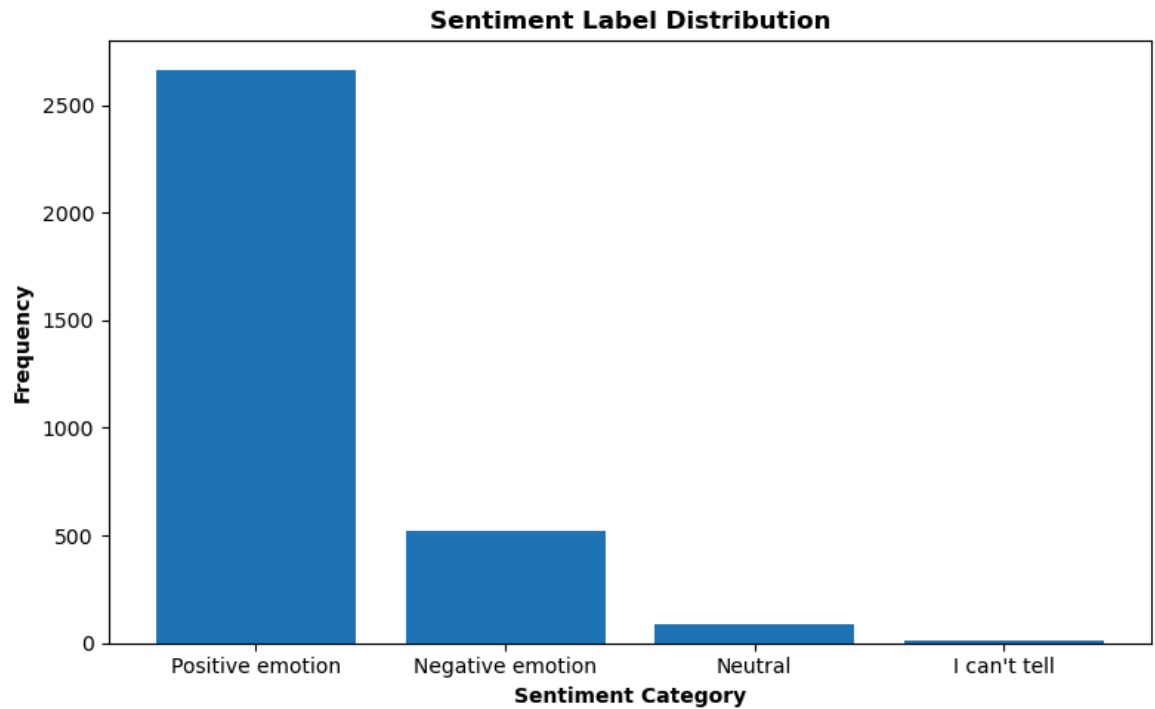
5: Exploratory Data Analysis (EDA)

5.1: Univariate Analysis

5.1.1: Sentiment Distribution

```
In [25]: sentiment_counts = df['sentiment_label'].value_counts()

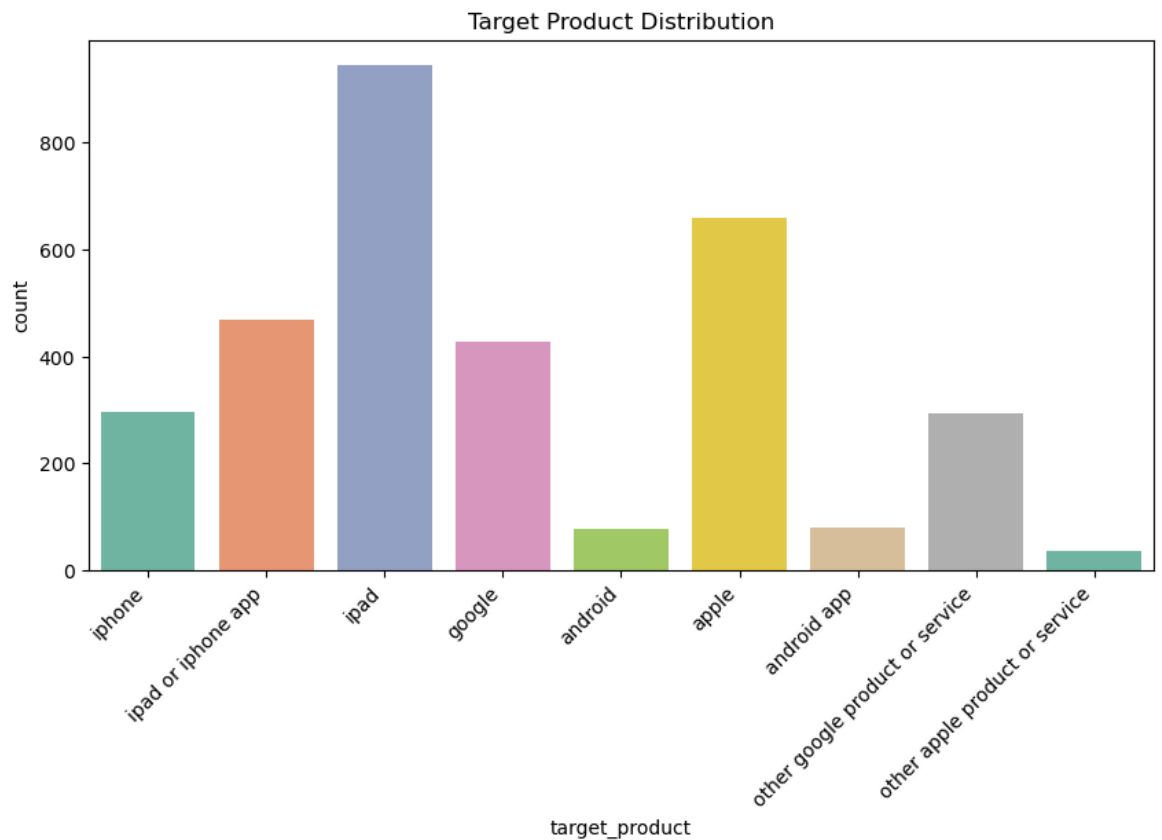
# Plot
plt.figure(figsize=(8,5))
plt.bar(sentiment_counts.index, sentiment_counts.values)
plt.title("Sentiment Label Distribution", fontweight='bold')
plt.xlabel("Sentiment Category", fontweight='bold')
plt.ylabel("Frequency", fontweight='bold')
plt.tight_layout()
plt.show()
```



The chart shows that **Positive emotion** is the most common sentiment in the dataset, followed by **Negative emotion**. The **Neutral** category appears much less frequently, and **I can't tell** is the least represented. This indicates an imbalanced distribution, with most tweets expressing positive reactions toward the products or brands. This **I can't tell** will either be dropped or merged with **Neutral** in later stages.

5.1.2: Target Product Distribution

```
In [26]: ▶ plt.figure(figsize=(10,5))
sns.countplot(data=df, x='target_product', palette="Set2")
plt.title("Target Product Distribution")
plt.xticks(rotation=45, ha='right')
plt.show()
```

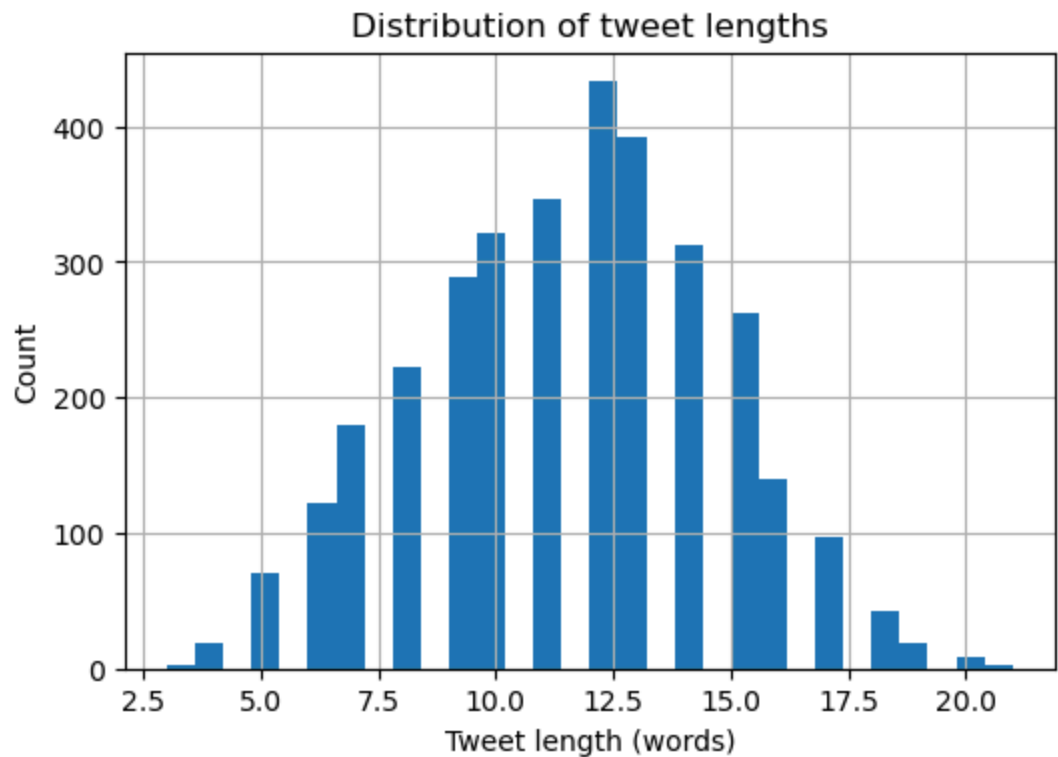


The chart shows that **iPad** is the most frequently mentioned product in the dataset, followed by **Apple**, **iPad or iPhone App**, and **Google**. Mentions of **Android**, **Android App**, and other Apple/Google services appear significantly less often. This indicates that user discussions are heavily centered around Apple's core products, with fewer tweets referencing Android-related items or secondary services.

5.1.3: Distribution of tweet lengths

```
In [27]: ▶ df['text_len'] = df['processed_text'].apply(lambda x: len(x.split()))
```

```
In [28]: #plotting the distribution of tweet lengths  
plt.figure(figsize=(6,4))  
df["text_len"].hist(bins=30)  
plt.xlabel("Tweet length (words)")  
plt.ylabel("Count")  
plt.title("Distribution of tweet lengths")  
plt.show()
```



The histogram shows that most tweets in the dataset contain between **10 and 15 words**, indicating that users typically write short, concise messages when expressing opinions about tech products. Very short or very long tweets are less common, and the overall distribution appears roughly normal, centered around medium-length tweets.

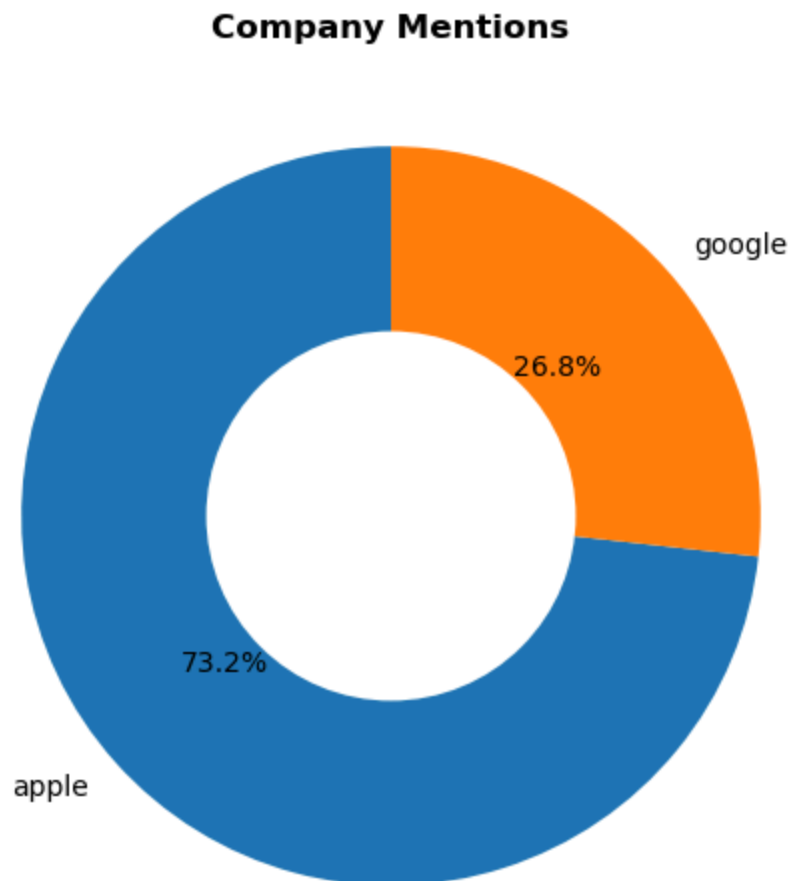
5.1.4: Distribution of Company

```
In [29]: ▶ company_counts = df['company'].value_counts()

plt.figure(figsize=(6,6))
plt.pie(company_counts, labels=company_counts.index, autopct='%1.1f%%', start

centre_circle = plt.Circle((0,0), 0.50, fc='white')
fig = plt.gcf()
fig.gca().add_artist(centre_circle)

plt.title("Company Mentions", fontweight='bold')
plt.show()
```

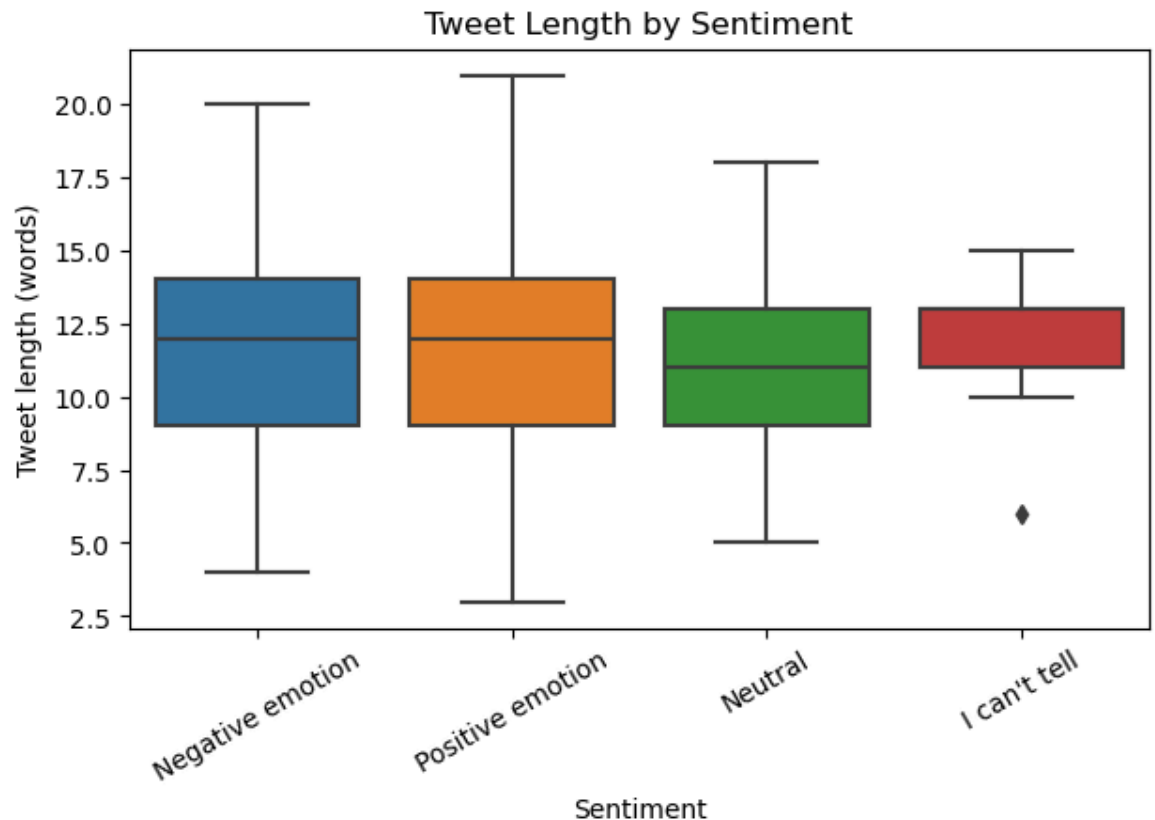


The donut chart shows that **Apple** is mentioned far more frequently than **Google** in the dataset. Apple accounts for about **73%** of all tweets, while Google represents only **27%**. This indicates that user discussions are heavily Apple-focused, with significantly fewer tweets referencing Google products or services.

5.2: Bivariate Analysis

5.2.1: Tweet Lengths by Sentiment Comparison

```
In [30]: ▶ #plotting the distribution of tweet lengths by sentiment
plt.figure(figsize=(7,4))
sns.boxplot(data=df, x="sentiment_label", y="text_len")
plt.title("Tweet Length by Sentiment")
plt.xlabel("Sentiment")
plt.ylabel("Tweet length (words)")
plt.xticks(rotation=30)
plt.show()
```



The boxplot shows that tweet lengths are **fairly similar** across all sentiment categories. **Positive** and **negative** tweets have comparable **median lengths**, while **neutral** tweets tend to be **slightly shorter on average**. The “I can't tell” category shows the **least variation**, indicating more consistency in tweet length. Overall, sentiment does not appear to drastically influence how long users' messages are.

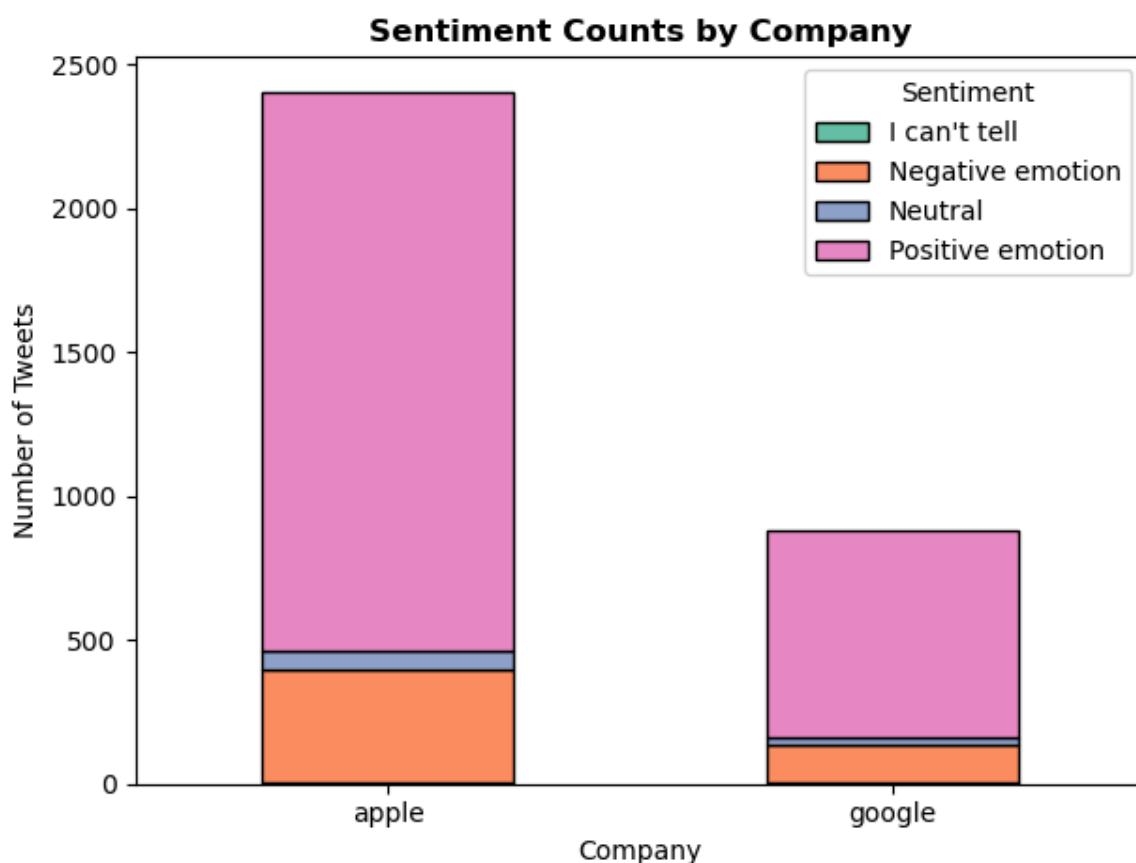
5.2.2: Sentiment Counts by Company

```
In [31]: # Data Prep
sent_by_company = df.groupby(["company", "sentiment_label"]).size().unstack()
colors = sns.color_palette("Set2", n_colors=len(sent_by_company.columns))

# Plotting
plt.figure(figsize=(10,8))
sent_by_company.plot(kind="bar", stacked=True, color=colors, edgecolor="black")

plt.title("Sentiment Counts by Company", fontweight='bold')
plt.xlabel("Company")
plt.ylabel("Number of Tweets")
plt.xticks(rotation=0)
plt.legend(title="Sentiment", bbox_to_anchor=(.65, 1), loc="upper left")
plt.tight_layout()
plt.show();
```

<Figure size 1000x800 with 0 Axes>



Building on the earlier sentiment distribution analysis which showed that **positive emotion** is the most dominant sentiment overall, this stacked bar chart further reveals how these sentiments are distributed between the two companies. Apple receives a much larger volume of tweets than Google, and this higher activity includes a substantial number of positive messages, reinforcing the earlier observation that users tend to express favorable opinions more frequently.

Negative and neutral sentiments remain comparatively lower for both companies, consistent with the general sentiment imbalance identified earlier. The "I can't tell" category is minimal across the board, indicating that most tweets express a clear emotional tone. Overall, this visualization

confirms that not only are users more inclined to discuss Apple, but these discussions are also

```
In [32]: # Creating pivot table
sent_prod = df.groupby(["target_product", "sentiment_label"]).size().unstack()

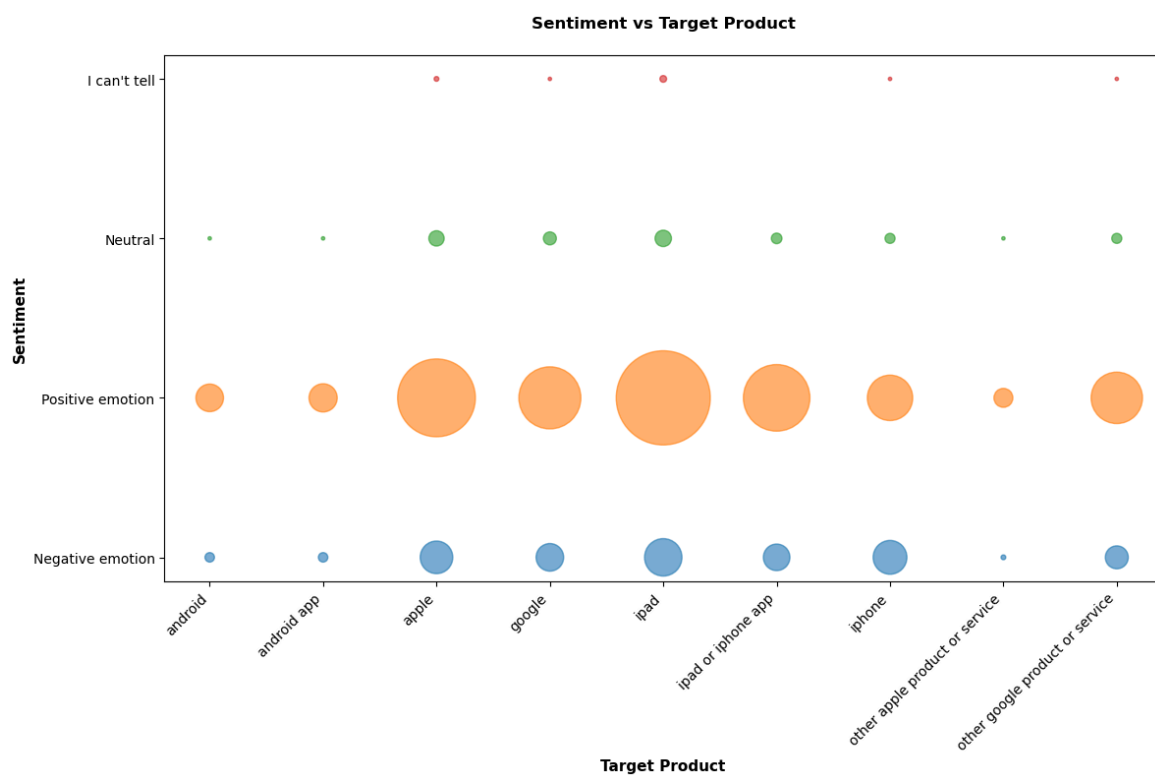
ordered_cols = ["Negative emotion", "Positive emotion", "Neutral", "I can't t
sent_prod = sent_prod[ordered_cols]

sent_prod_reset = sent_prod.reset_index()

fig, ax = plt.subplots(figsize=(12,8))

for sentiment in ordered_cols:
    ax.scatter(
        sent_prod_reset["target_product"],
        [sentiment] * len(sent_prod_reset),
        s=sent_prod_reset[sentiment] * 6,
        alpha=0.6,
        label=sentiment
    )
ax.set_title("Sentiment vs Target Product", fontsize=12, fontweight='bold', p
ax.set_xlabel("Target Product", fontsize=11, fontweight='bold')
ax.set_ylabel("Sentiment", fontsize=11, fontweight='bold')
plt.xticks(rotation=45, ha="right")

plt.tight_layout()
plt.show()
```



The bubble chart provides a visual comparison of how different sentiments are distributed across product categories. Larger bubbles represent a higher number of tweets. Positive sentiment dominates across nearly all products, especially for **iPad**, **iPhone**, and **Apple** categories, where large bubbles indicate strong positive engagement. Negative sentiment bubbles appear

moderately across several products but remain noticeably smaller, while neutral and "I can't tell" sentiments show minimal activity. This visualization highlights that most user discussions across product types tend to express positive emotions, with relatively few negative or unclear sentiments.

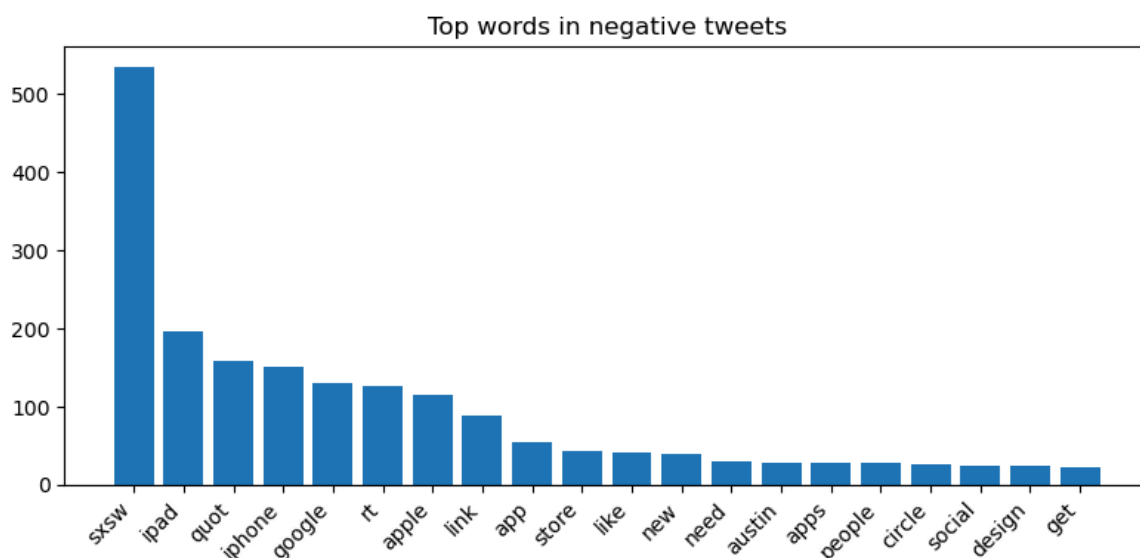
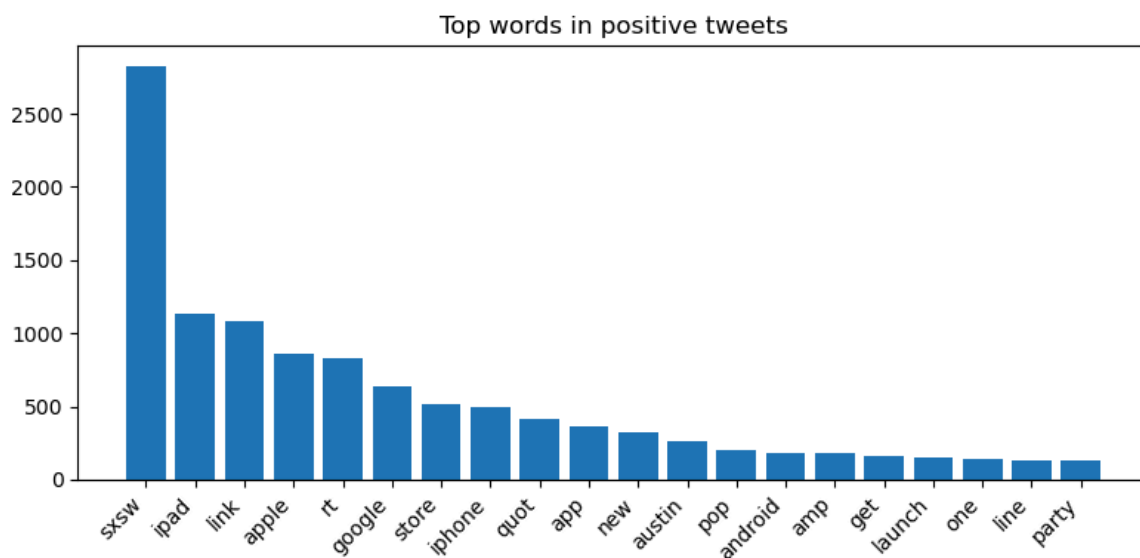
5.3: NLP Specific Visuals

5.3.1: Top Words Across Sentiments

```
In [33]: ▶ def get_top_n_words(corpus, n=20):  
        vec = CountVectorizer()  
        X = vec.fit_transform(corpus)  
        sum_words = X.sum(axis=0)  
        words_freq = [(word, int(sum_words[0, idx])) for word, idx in vec.vocabulary_.items()]  
        words_freq = sorted(words_freq, key=lambda x: x[1], reverse=True)  
        return words_freq[:n]  
  
In [34]: ▶ pos_corpus = df[df["sentiment_label"] == "Positive emotion"]["processed_text"]  
        neg_corpus = df[df["sentiment_label"] == "Negative emotion"]["processed_text"]  
  
        top_pos = get_top_n_words(pos_corpus, n=20)  
        top_neg = get_top_n_words(neg_corpus, n=20)
```

```
In [35]: def plot_top_words(word_freq, title):
        words, freqs = zip(*word_freq)
        plt.figure(figsize=(8,4))
        plt.bar(range(len(words)), freqs)
        plt.xticks(range(len(words)), words, rotation=45, ha="right")
        plt.title(title)
        plt.tight_layout()
        plt.show()

        plot_top_words(top_pos, "Top words in positive tweets")
        plot_top_words(top_neg, "Top words in negative tweets")
```



The charts compare the most frequently used words in positive and negative tweets. Positive tweets prominently feature terms like **sxsw**, **ipad**, **link**, and **apple**, suggesting excitement around events, products, and updates. Negative tweets also include some of the same product related words, such as **ipad**, **iphone**, and **store**, but often in different contexts. Overall, positive tweets contain a broader variety of high-frequency terms, while negative tweets show fewer but more concentrated word patterns.

5.3.2: Top Bigrams

```
In [36]: ▶ def get_top_n_ngrams(corpus, n=20, ngram_range=(2,2)):
    vec = CountVectorizer(ngram_range=ngram_range)
    X = vec.fit_transform(corpus)
    sum_words = X.sum(axis=0)
    words_freq = [(word, int(sum_words[0, idx])) for word, idx in vec.vocabulary_.items()]
    words_freq = sorted(words_freq, key=lambda x: x[1], reverse=True)
    return words_freq[:n]

    top_pos_bi = get_top_n_ngrams(pos_corpus, n=20, ngram_range=(2,2))
    top_neg_bi = get_top_n_ngrams(neg_corpus, n=20, ngram_range=(2,2))

    df_pos_bigrams = pd.DataFrame(top_pos_bi, columns=["bigram", "frequency"])
    df_neg_bigrams = pd.DataFrame(top_neg_bi, columns=["bigram", "frequency"])
```

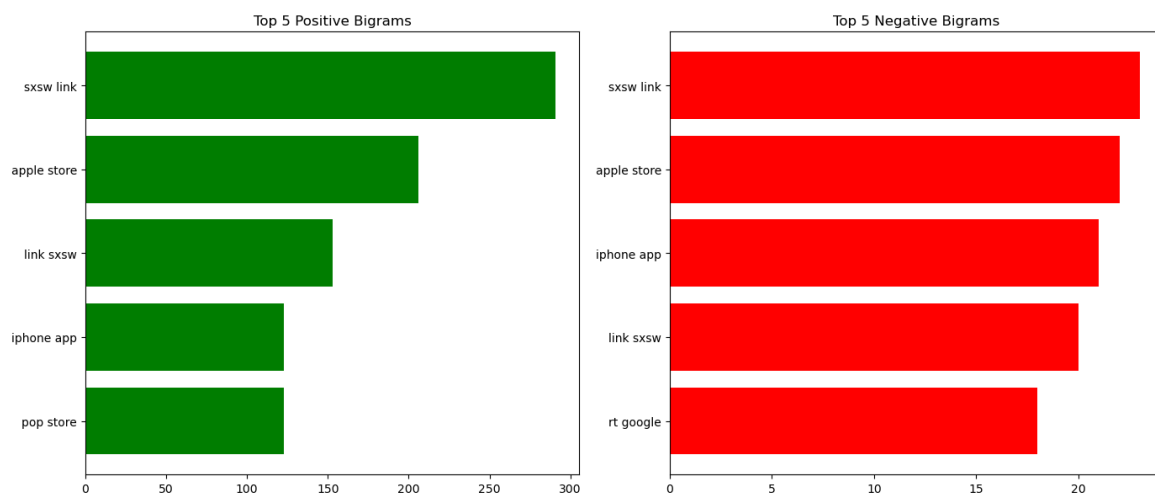
```
In [37]: ▶ #Visualization
pos5 = df_pos_bigrams.head()
neg5 = df_neg_bigrams.head()

fig, axes = plt.subplots(1, 2, figsize=(14,6))

axes[0].barh(pos5['bigram'], pos5['frequency'], color='green')
axes[0].set_title("Top 5 Positive Bigrams")
axes[0].invert_yaxis()

axes[1].barh(neg5['bigram'], neg5['frequency'], color='red')
axes[1].set_title("Top 5 Negative Bigrams")
axes[1].invert_yaxis()

plt.tight_layout()
plt.show()
```



The most frequent bigrams in positive tweets highlight product-related excitement and event-driven discussions. The top examples are as shown above. These bigrams suggest strong engagement around SXSW events, Apple product releases, and app-related interactions. On the other hand, Negative tweets show fewer repeated patterns, but still reflect concerns or issues linked to similar

products and services. The top examples are as shown above. These patterns indicate that negative sentiment often revolves around the same tech ecosystems but appears far. The

5.3.3: Word Cloud

The word clouds illustrate the most frequently occurring terms in positive and negative tweets. In **negative reviews**, words such as *ipad*, *iphone*, *apple*, and *sxsw* appear prominently, often associated with issues, complaints, or user frustration.

In contrast, **positive reviews** feature similar product related terms, but within a more favorable context, with words like *awesome*, *cool*, *link*, and *great* appearing more often.

Both word clouds highlight that discussions around **Apple products** and the **SXSW** event dominate user conversations, but the tone and surrounding context differ significantly between positive and negative sentiments.

```
In [39]: keywords = ["battery", "crash", "lag", "slow", "overheat", "bug", "update"]

for kw in keywords:
    total = df["processed_text"].str.contains(kw, case=False, na=False).sum()
    neg = df[(df["sentiment_label"]=="Negative emotion") & (df["processed_text"].str.contains(kw, case=False, na=False))].sum()
    print(f"{kw}: total={total}, negative={neg}")

battery: total=34, negative=20
crash: total=15, negative=11
lag: total=2, negative=1
slow: total=8, negative=1
overheat: total=1, negative=1
bug: total=5, negative=3
update: total=43, negative=4
```


6: Feature Engineering

6.1: TF-IDF vectors

Feature engineering is critical for transforming raw text data into meaningful numerical representations that machine learning models can understand. In this section, we create multiple types of features:

1. **TF-IDF Vectors:** Capture word importance across the corpus
2. **Numeric Features:** Text statistics and sentiment indicators
3. **Advanced Features:** N-grams, special characters, and linguistic patterns
4. **Target Encoding:** Convert sentiment labels to numerical format

These engineered features will be used to train and evaluate our sentiment classification models.

```
In [40]:  # Initialize TF-IDF Vectorizer
# max_features limits vocabulary size, ngram_range includes unigrams and bigrams
tfidf = TfidfVectorizer(max_features=5000, ngram_range=(1, 2), min_df=2, max_df=10)

# Fit and transform the processed text
X_tfidf = tfidf.fit_transform(df['processed_text'])

print(f"TF-IDF matrix shape: {X_tfidf.shape}")
print(f"Vocabulary size: {len(tfidf.vocabulary_)}")
print(f"\nSample features (first 20):")
print(tfidf.get_feature_names_out()[:20])
```

TF-IDF matrix shape: (3282, 5000)

Vocabulary size: 5000

Sample features (first 20):

```
['aapl' 'able' 'absolutely' 'abt' 'access' 'accessible' 'accessory' 'aclu'
 'aclu google' 'aclu party' 'acquired' 'across' 'action' 'action link'
 'actsofsharing' 'actsofsharing com' 'actual' 'actually' 'ad' 'add']
```

6.2: Additional numeric features

```
In [41]: # Create additional numeric features that can improve model performance

# 1. Text length features
df['char_count'] = df['processed_text'].apply(len)
df['word_count'] = df['processed_text'].apply(lambda x: len(x.split()))
df['avg_word_length'] = df['processed_text'].apply(lambda x: np.mean([len(wor

# 2. Special character counts
df['exclamation_count'] = df['tweet'].apply(lambda x: x.count('!'))
df['question_count'] = df['tweet'].apply(lambda x: x.count('?'))
df['uppercase_count'] = df['tweet'].apply(lambda x: sum(1 for c in x if c.isu
df['hashtag_count'] = df['tweet'].apply(lambda x: x.count('#'))
df['mention_count'] = df['tweet'].apply(lambda x: x.count('@'))

# 3. Sentiment-related features
df['positive_word_count'] = df['processed_text'].apply(lambda x: sum(1 for wc
df['negative_word_count'] = df['processed_text'].apply(lambda x: sum(1 for wc

# 4. Company encoding (binary features)
df['is_apple'] = (df['company'] == 'apple').astype(int)
df['is_google'] = (df['company'] == 'google').astype(int)

# Display the new features
print("New numeric features created:")
print(df[['char_count', 'word_count', 'avg_word_length', 'exclamation_count',
          'question_count', 'positive_word_count', 'negative_word_count',
          'is_apple', 'is_google']].head(10))

print("\n" + "="*50)
print("Feature Statistics:")
print(df[['char_count', 'word_count', 'avg_word_length', 'exclamation_count',
          'question_count', 'positive_word_count', 'negative_word_count']].de
```

New numeric features created:

	char_count	word_count	avg_word_length	exclamation_count	question_count \
0	70	12	4.916667	1	
0					
1	77	13	5.000000	0	
1					
2	24	5	4.000000	0	
0					
3	46	8	4.875000	0	
0					
4	102	16	5.437500	0	
0					
5	77	13	5.000000	0	
0					
6	63	10	5.400000	1	
0					
7	67	11	5.181818	0	
0					
8	63	10	5.400000	0	
0					
9	61	11	4.636364	0	
0					

	positive_word_count	negative_word_count	is_apple	is_google
0	0	0	1	0
1	1	0	1	0
2	0	0	1	0
3	0	0	1	0
4	1	0	0	1
5	1	0	0	1
6	0	0	1	0
7	0	0	1	0
8	0	0	0	1
9	0	0	0	1

=====
Feature Statistics:

	char_count	word_count	avg_word_length	exclamation_count \
count	3282.000000	3282.000000	3282.000000	3282.000000
mean	68.974101	11.475320	5.102863	0.374771
std	20.103290	3.143257	0.693427	0.777431
min	14.000000	3.000000	3.400000	0.000000
25%	54.250000	9.000000	4.615385	0.000000
50%	70.000000	12.000000	5.000000	0.000000
75%	83.000000	14.000000	5.500000	1.000000
max	134.000000	21.000000	8.125000	9.000000

	question_count	positive_word_count	negative_word_count
count	3282.000000	3282.000000	3282.000000
mean	0.127666	0.166971	0.012797
std	0.403999	0.403618	0.120274
min	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000

75%	0.000000	0.000000	0.000000
max	4.000000	3.000000	2.000000

```
In [42]: # Visualize some of the numeric features

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

# 1. Word count distribution by sentiment
sns.boxplot(data=df,x="sentiment_label",y="word_count",palette="Set2",ax=axes[0,0])
axes[0,0].set_title("Word Count by Sentiment", fontsize=13, fontweight='bold')
axes[0,0].set_xlabel("Sentiment")
axes[0,0].set_ylabel("Word Count")
axes[0,0].tick_params(axis='x', rotation=45)

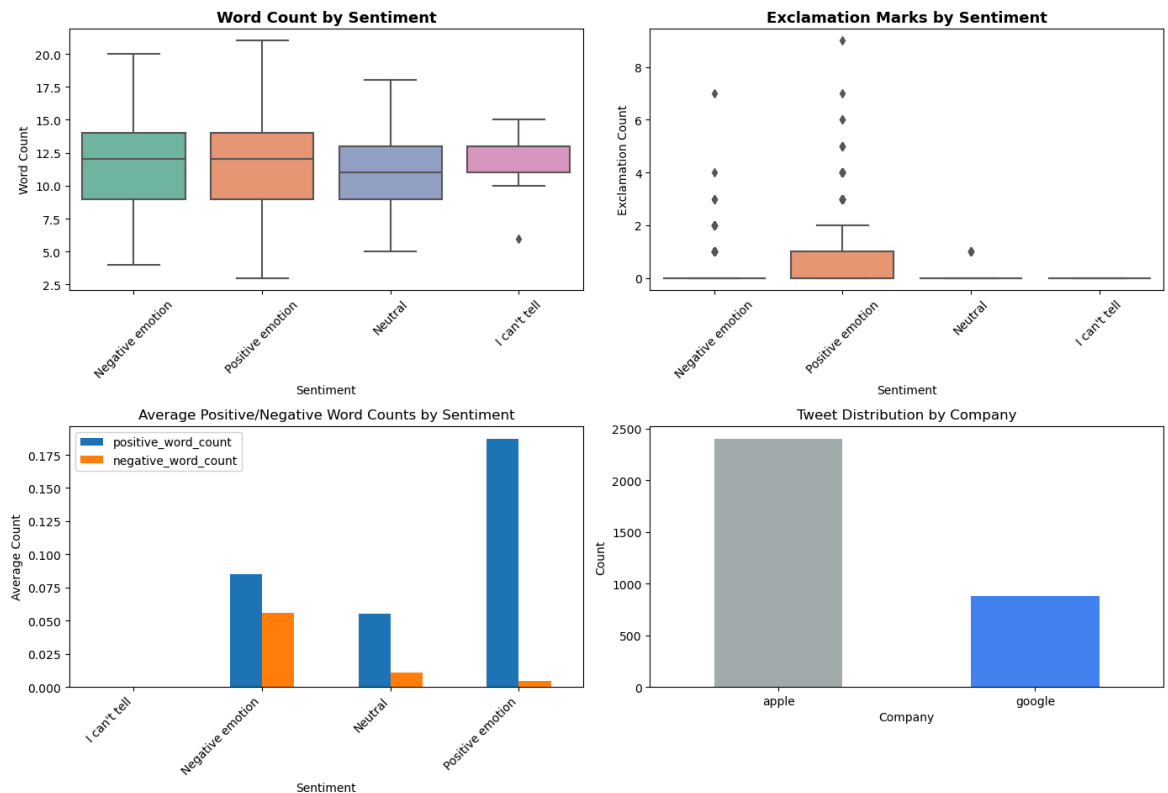
# 2. Exclamation marks by sentiment
sns.boxplot(data=df,x="sentiment_label",y="exclamation_count",palette="Set2",ax=axes[0,1])
axes[0,1].set_title("Exclamation Marks by Sentiment", fontsize=13, fontweight='bold')
axes[0,1].set_xlabel("Sentiment")
axes[0,1].set_ylabel("Exclamation Count")
axes[0,1].tick_params(axis='x', rotation=45)

# 3. Positive word count by sentiment
sentiment_counts = df.groupby('sentiment_label')[['positive_word_count', 'negative_word_count']]
sentiment_counts.agg(['sum']).reset_index().plot(kind='bar', ax=axes[1, 0])
axes[1, 0].set_title('Average Positive/Negative Word Counts by Sentiment')
axes[1, 0].set_xlabel('Sentiment')
axes[1, 0].set_ylabel('Average Count')
axes[1, 0].set_xticklabels(axes[1, 0].get_xticklabels(), rotation=45, ha='right')

# 4. Company distribution
df['company'].value_counts().plot(kind='bar', ax=axes[1, 1], color=['#A2A9AD'])
axes[1, 1].set_title('Tweet Distribution by Company')
axes[1, 1].set_xlabel('Company')
axes[1, 1].set_ylabel('Count')
axes[1, 1].set_xticklabels(axes[1, 1].get_xticklabels(), rotation=0)

plt.suptitle('Numeric Feature Analysis', fontsize=16, y=1.00)
plt.tight_layout()
plt.show()
```

Numeric Feature Analysis



6.3: Encoded target labels

```
In [43]: df['sentiment_label'].value_counts()
```

```
Out[43]: sentiment_label
Positive emotion    2664
Negative emotion    518
Neutral             91
I can't tell        9
Name: count, dtype: int64
```

Because the *"I can't tell"* category has only 9 samples, it is too small for reliable model training. Such a tiny class causes problems during cross-validation, prevents oversampling methods from working, and leads to unstable or undefined performance metrics.

To avoid these issues and improve overall model stability, we merge *"I can't tell"* into the **Neutral** class. This simplifies the sentiment structure and ensures that ambiguous or unclear expressions are handled consistently.

```
In [44]: ▶ # Merging the very small "I can't tell" class into Neutral
df['sentiment_label'] = df['sentiment_label'].replace({
    "I can't tell": "Neutral"
})

# Check updated distribution
print("\n" + "="*50)
print(df['sentiment_label'].value_counts())
print("\n" + "="*50)
```

```
=====
sentiment_label
Positive emotion    2664
Negative emotion    518
Neutral             100
Name: count, dtype: int64
=====
```

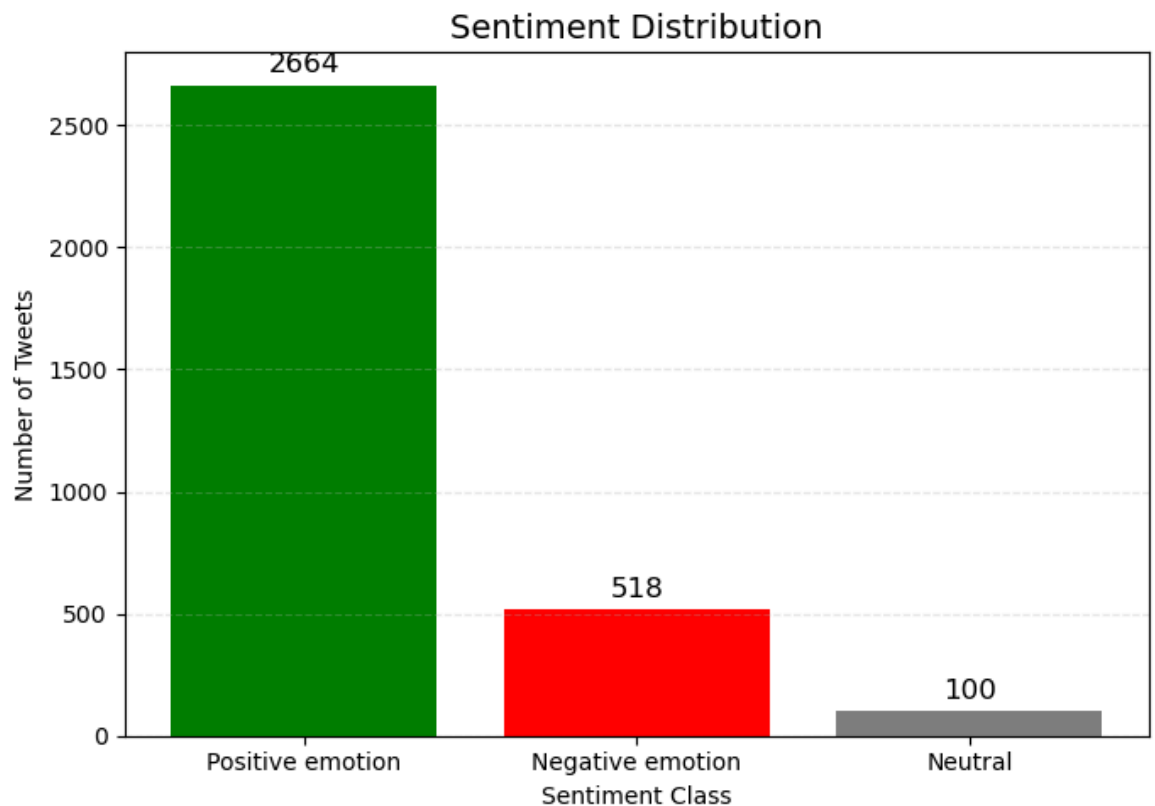
In [93]:

```
# Updated value counts
sent_counts = df['sentiment_label'].value_counts()

# Plot
plt.figure(figsize=(7,5))
bars = plt.bar(sent_counts.index, sent_counts.values, color=['green','red','g

for bar in bars:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, height + 30,
             f"{height}", ha='center', va='bottom', fontsize=12)

plt.title("Sentiment Distribution", fontsize=14)
plt.ylabel("Number of Tweets")
plt.xlabel("Sentiment Class")
plt.grid(axis='y', linestyle='--', alpha=0.3)
plt.tight_layout()
plt.show()
```



```

In [45]: # Encode sentiment labels for machine Learning models
# Convert categorical sentiment labels to numeric values

# Display unique sentiment labels
print("\n" + "="*50)
print("Unique sentiment labels:")
print(df['sentiment_label'].unique())
print("\n" + "="*50)
print("\nLabel distribution:")
print(df['sentiment_label'].value_counts())
print("\n" + "="*50)

# Initialize Label Encoder
label_encoder = LabelEncoder()

# Creating a copy for multiclass classification
df_mclass = df.copy()

# Fit and transform the sentiment labels
df_mclass['sentiment_encoded'] = label_encoder.fit_transform(df_mclass['sentiment_label'])

# Creating a binary df for Logistic Regression
df_binary = df[df['sentiment_label'].isin(['Positive emotion', 'Negative emotion'])]
print("Binary sentiment dataset created!")
print(df_binary['sentiment_label'].value_counts())
label_encoder_binary = LabelEncoder()
df_binary['sentiment_encoded'] = label_encoder_binary.fit_transform(df_binary['sentiment_label'])
print("\n" + "="*50)
print("Binary dataset encoding:")
print(df_binary[['sentiment_label', 'sentiment_encoded']].head())

# Create mapping for reference
label_mapping = dict(zip(label_encoder.classes_, label_encoder.transform(label_encoder.classes_)))
print("\n" + "="*50)
print("Label Encoding Mapping:")
for label, code in label_mapping.items():
    print(f" {label} -> {code}")

# Display examples
print("\n" + "="*50)
print("Sample of Multi_Class encoded labels:")
print(df_mclass[['sentiment_label', 'sentiment_encoded']].head(10))

```

```

=====
Unique sentiment labels:
['Negative emotion' 'Positive emotion' 'Neutral']
=====

Label distribution:
sentiment_label
Positive emotion    2664
Negative emotion    518
Neutral             100
Name: count, dtype: int64

=====

Binary sentiment dataset created!
sentiment_label
Positive emotion    2664
Negative emotion    518
Name: count, dtype: int64

=====

Binary dataset encoding:
      sentiment_label  sentiment_encoded
0  Negative emotion           0
1  Positive emotion           1
2  Positive emotion           1
3  Negative emotion           0
4  Positive emotion           1

=====

Label Encoding Mapping:
  Negative emotion -> 0
    Neutral -> 1
  Positive emotion -> 2

=====

Sample of Multi_Class encoded labels:
      sentiment_label  sentiment_encoded
0  Negative emotion           0
1  Positive emotion           2
2  Positive emotion           2
3  Negative emotion           0
4  Positive emotion           2
5  Positive emotion           2
6  Positive emotion           2
7  Positive emotion           2
8  Positive emotion           2
9  Positive emotion           2

```

6.4: Advanced Text Features

Beyond basic TF-IDF and simple counts, we can extract more sophisticated linguistic features that capture nuances in the text.

```

In [46]: # Advanced linguistic features
def add_advanced_features(df):

    # 1. Punctuation density
    df['punctuation_count'] = df['tweet'].apply(lambda x: sum(1 for c in x if
    df['punctuation_density'] = df['punctuation_count'] / (df['char_count'] + 1)

    # 2. Capital letter ratio (intensity indicator)
    df['capital_ratio'] = df['uppercase_count'] / (df['char_count'] + 1)

    # 3. Unique word ratio (vocabulary diversity)
    df['unique_word_ratio'] = df['processed_text'].apply(lambda x: len(set(x.

    # 4. Repeated character patterns (e.g., "soooo" or "!!!")
    df['repeated_chars'] = df['tweet'].apply(lambda x: len(re.findall(r'(\.)\1

    # 5. URL presence (already removed but check original)
    df['has_url'] = df['tweet'].apply(lambda x: 1 if 'http' in x.lower() or '

    # 6. Ellipsis count (indicates trailing thought)
    df['ellipsis_count'] = df['tweet'].apply(lambda x: x.count('...'))

    # 7. Average sentence length (approximated by splitting on periods)
    df['sentence_count'] = df['tweet'].apply(lambda x: len([s for s in x.spli
    df['avg_sentence_length'] = df['word_count'] / (df['sentence_count'] + 1)

    return df

#Applying to multiclass dataset
df_mclass = add_advanced_features(df_mclass)

#Applying to binary df
df_binary = add_advanced_features(df_binary)

```

```
In [47]: #Visuals
print("\nSample of advanced features:")
cols = ['punctuation_density', 'capital_ratio', 'unique_word_ratio',
        'repeated_chars', 'has_url', 'ellipsis_count']
print(df_mclass[cols].head(10))

print("\n" + "="*50)
print("Advanced Feature Statistics:")
print(df_mclass[cols].describe())
```

Sample of advanced features:

	punctuation_density	capital_ratio	unique_word_ratio	repeated_chars	\
0	0.084507	0.211268	0.923077	0	
1	0.038462	0.128205	0.928571	0	
2	0.080000	0.280000	0.833333	0	
3	0.021277	0.042553	0.777778	0	
4	0.029126	0.135922	0.941176	0	
5	0.025641	0.102564	0.928571	0	
6	0.046875	0.109375	0.909091	0	
7	0.000000	0.044118	0.916667	0	
8	0.031250	0.093750	0.909091	1	
9	0.064516	0.258065	0.916667	0	

	has_url	ellipsis_count
0	0	0
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0
6	1	0
7	0	0
8	0	0
9	1	0

=====

Advanced Feature Statistics:

	punctuation_density	capital_ratio	unique_word_ratio	repeated_chars	\
count	3282.000000	3282.000000	3282.000000	3282.00000	
0					
mean	0.035856	0.112505	0.879902	0.11029	
9					
std	0.026884	0.073461	0.055743	0.35353	
1					
min	0.000000	0.000000	0.600000	0.00000	
0					
25%	0.018182	0.059406	0.857143	0.00000	
0					
50%	0.031746	0.102273	0.900000	0.00000	
0					
75%	0.048387	0.149174	0.923077	0.00000	
0					
max	0.214286	0.814815	0.950000	4.00000	
0					

	has_url	ellipsis_count
count	3282.000000	3282.000000
mean	0.006703	0.071603
std	0.081611	0.275026
min	0.000000	0.000000
25%	0.000000	0.000000
50%	0.000000	0.000000
75%	0.000000	0.000000
max	1.000000	3.000000

6.5: Combining Features for Model Training

Now we'll combine TF-IDF features with our numeric features to create comprehensive feature sets for different modeling approaches.

```
In [48]: ▶ def prepare_features(df, tfidf_vectorizer=None, scaler=None):
# 1. TF-IDF
if tfidf_vectorizer is None:
    tfidf_vectorizer = TfidfVectorizer(
        max_features=5000,
        ngram_range=(1, 2),
        min_df=2,
        max_df=0.95
    )
    X_tfidf = tfidf_vectorizer.fit_transform(df['processed_text'])
else:
    X_tfidf = tfidf_vectorizer.transform(df['processed_text'])

# 2. Numeric Features
numeric_features = [
    'char_count', 'word_count', 'avg_word_length',
    'exclamation_count', 'question_count', 'uppercase_count',
    'hashtag_count', 'mention_count',
    'positive_word_count', 'negative_word_count',
    'is_apple', 'is_google',
    'punctuation_density', 'capital_ratio', 'unique_word_ratio',
    'repeated_chars', 'has_url', 'ellipsis_count', 'avg_sentence_length'
]

X_numeric = df[numeric_features].values

# 3. Standardization
if scaler is None:
    scaler = StandardScaler()
    X_numeric_scaled = scaler.fit_transform(X_numeric)
else:
    X_numeric_scaled = scaler.transform(X_numeric)

# 4. Combine Features
X_tfidf_dense = X_tfidf.toarray()
X_combined = np.hstack([X_tfidf_dense, X_numeric_scaled])

# 5. Labels
y = df['sentiment_encoded'].values

return X_tfidf, X_combined, y, tfidf_vectorizer, scaler
```

```
In [49]: ▶ #Multiclass dataset
X_tfidf_mclass, X_combined_mclass, y_mclass, tfidf_vec, scaler = prepare_feat
print(X_combined_mclass.shape, y_mclass.shape)

(3282, 5019) (3282,)
```

```
In [50]: #Binary dataset
X_tfidf_binary, X_combined_binary, y_binary, _, _ = prepare_features(
    df_binary,
    tfidf_vectorizer=tfidf_vec,
    scaler=scaler
)
print(X_combined_binary.shape, y_binary.shape)

(3182, 5019) (3182,)
```

```
In [51]: #Label mappings
label_mapping_mclass = dict(zip(label_encoder.classes_,
                                label_encoder.transform(label_encoder.classes_)))

label_mapping_binary = dict(zip(label_encoder_binary.classes_,
                                label_encoder_binary.transform(label_encoder_
```

```
In [52]: def print_feature_summary(name, X_tfidf, X_combined, y, label_mapping):
    print(f"\nFeature Engineering Summary for: {name}")
    print("=" * 70)

    print(f"TF-IDF features shape:      {X_tfidf.shape}")
    print(f"Combined features shape:      {X_combined.shape}")
    print(f"Target variable shape:          {y.shape}")

    print("\nNumber of classes:", len(np.unique(y)))
    print("Class distribution:")

    for label, code in label_mapping.items():
        count = np.sum(y == code)
        if count > 0:
            percentage = (count / len(y)) * 100
            print(f" {label} (encoded {code}): {count} samples ({percentage}%)")

    print("=" * 70)
    print("Ready for model training!")
    print("\n")
```

```
In [53]: ▶ print_feature_summary(
            name="Binary Dataset",
            X_tfidf=X_tfidf_binary,
            X_combined=X_combined_binary,
            y=y_binary,
            label_mapping=label_mapping_binary
        )
```

Feature Engineering Summary for: Binary Dataset

```
=====
TF-IDF features shape:      (3182, 5000)
Combined features shape:    (3182, 5019)
Target variable shape:      (3182,)
```

Number of classes: 2

Class distribution:

Negative emotion (encoded 0): 518 samples (16.28%)

Positive emotion (encoded 1): 2664 samples (83.72%)

```
=====
Ready for model training!
```

```
In [54]: ▶ print_feature_summary(
            name="Multiclass Dataset",
            X_tfidf=X_tfidf_mclass,
            X_combined=X_combined_mclass,
            y=y_mclass,
            label_mapping=label_mapping_mclass
        )
```

Feature Engineering Summary for: Multiclass Dataset

```
=====
TF-IDF features shape:      (3282, 5000)
Combined features shape:    (3282, 5019)
Target variable shape:      (3282,)
```

Number of classes: 3

Class distribution:

Negative emotion (encoded 0): 518 samples (15.78%)

Neutral (encoded 1): 100 samples (3.05%)

Positive emotion (encoded 2): 2664 samples (81.17%)

```
=====
Ready for model training!
```

6.6: Splitting the Data into Training and Testing sets

```
In [55]: ▶ def split_data(X, y, label_mapping, dataset_name="Dataset", test_size=0.2, random_state=None):
# Splitting using stratification
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=test_size,
    random_state=random_state,
    stratify=y
)

# Summary
print(f"\nTrain-Test Split Summary: {dataset_name}")
print("=" * 70)
print(f"Training set size: {X_train.shape[0]} samples")
print(f"Testing set size: {X_test.shape[0]} samples")
print(f"Feature dimensions: {X_train.shape[1]}")

print("\nClass distribution in TRAIN set:")
for label, code in label_mapping.items():
    count = np.sum(y_train == code)
    if count > 0:
        pct = (count / len(y_train)) * 100
        print(f"  {label}: {count} samples ({pct:.2f}%)")

print("\nClass distribution in TEST set:")
for label, code in label_mapping.items():
    count = np.sum(y_test == code)
    if count > 0:
        pct = (count / len(y_test)) * 100
        print(f"  {label}: {count} samples ({pct:.2f}%)")

print("=" * 70)

return X_train, X_test, y_train, y_test
```

```
In [56]: #multiclass dataset combined
X_train_mclass, X_test_mclass, y_train_mclass, y_test_mclass = split_data(
    X_combined_mclass,
    y_mclass,
    label_mapping_mclass,
    dataset_name="Multiclass Dataset"
)
```

Train-Test Split Summary: Multiclass Dataset

```
=====
Training set size: 2625 samples
Testing set size: 657 samples
Feature dimensions: 5019
```

Class distribution in TRAIN set:

```
Negative emotion: 414 samples (15.77%)
Neutral: 80 samples (3.05%)
Positive emotion: 2131 samples (81.18%)
```

Class distribution in TEST set:

```
Negative emotion: 104 samples (15.83%)
Neutral: 20 samples (3.04%)
Positive emotion: 533 samples (81.13%)
=====
```

```
In [57]: #binary Dataset_combined
X_train_binary, X_test_binary, y_train_binary, y_test_binary = split_data(
    X_combined_binary,
    y_binary,
    label_mapping_binary,
    dataset_name="Binary Dataset"
)
```

Train-Test Split Summary: Binary Dataset

```
=====
Training set size: 2545 samples
Testing set size: 637 samples
Feature dimensions: 5019
```

Class distribution in TRAIN set:

```
Negative emotion: 414 samples (16.27%)
Positive emotion: 2131 samples (83.73%)
```

Class distribution in TEST set:

```
Negative emotion: 104 samples (16.33%)
Positive emotion: 533 samples (83.67%)
=====
```

```
In [58]: ▶ # Binary Dataset (TF-IDF only)
X_train_tfidf_binary, X_test_tfidf_binary, y_train_binary, y_test_binary = sp
    X_tfidf_binary,
    y_binary,
    label_mapping_binary,
    dataset_name="Binary Dataset (TF-IDF Only)"
)
```

Train-Test Split Summary: Binary Dataset (TF-IDF Only)

=====

Training set size: 2545 samples

Testing set size: 637 samples

Feature dimensions: 5000

Class distribution in TRAIN set:

Negative emotion: 414 samples (16.27%)

Positive emotion: 2131 samples (83.73%)

Class distribution in TEST set:

Negative emotion: 104 samples (16.33%)

Positive emotion: 533 samples (83.67%)

=====

```
In [59]: ▶ # Multiclass Dataset (TF-IDF only)
X_train_tfidf_mclass, X_test_tfidf_mclass, y_train_mclass, y_test_mclass = sp
    X_tfidf_mclass,
    y_mclass,
    label_mapping_mclass,
    dataset_name="Multiclass Dataset (TF-IDF Only)"
)
```

Train-Test Split Summary: Multiclass Dataset (TF-IDF Only)

=====

Training set size: 2625 samples

Testing set size: 657 samples

Feature dimensions: 5000

Class distribution in TRAIN set:

Negative emotion: 414 samples (15.77%)

Neutral: 80 samples (3.05%)

Positive emotion: 2131 samples (81.18%)

Class distribution in TEST set:

Negative emotion: 104 samples (15.83%)

Neutral: 20 samples (3.04%)

Positive emotion: 533 samples (81.13%)

=====

7: Baseline Models

7.1: Logistic Regression

7.1:1 Basic_log_regression with combined Features

```
In [60]: ▶ #confusion Matrix
def plot_confusion_matrix(cm, labels, title):
    plt.figure(figsize=(6,5))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
                xticklabels=labels, yticklabels=labels)
    plt.title(title)
    plt.ylabel("Actual")
    plt.xlabel("Predicted")
    plt.show()

In [61]: ▶ def run_basic_log_regression(X_train, X_test, y_train, y_test, model_name):
    print(f"\n{'='*70}")
    print(f"Running - {model_name}")
    print(f"{'='*70}")

    lr = LogisticRegression(class_weight="balanced", max_iter=2000, random_st
    lr.fit(X_train, y_train)

    y_pred = lr.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
    print(f"\nAccuracy: {accuracy:.4f}")
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred, target_names=['Negative', 'Po

    cm = confusion_matrix(y_test, y_pred)
    plot_confusion_matrix(cm, ['Negative', 'Positive'], f"{model_name} - Conf

    return lr, accuracy
```

```
In [62]: #Basic Logistic Regression with combined features
model_basic_combined, acc_basic_combined = run_basic_log_regression(
    X_train_binary, X_test_binary,
    y_train_binary, y_test_binary,
    model_name="Binary Logistic Regression (Combined Features)"
)
```

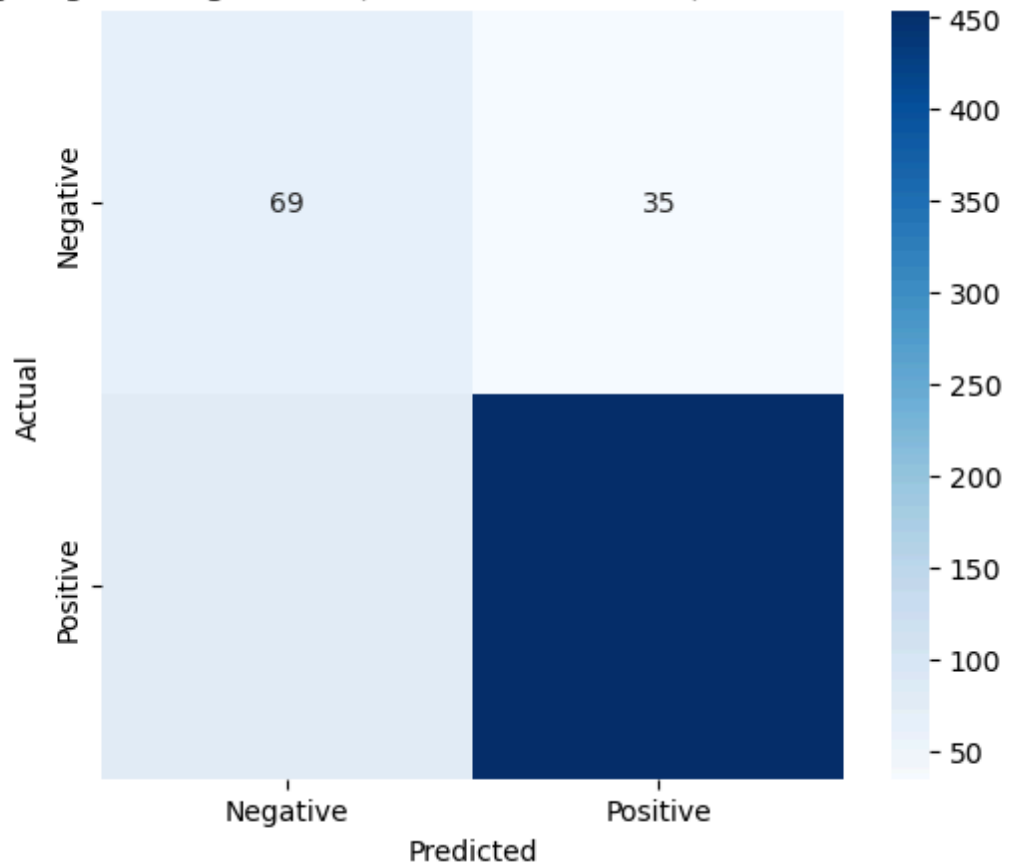
```
=====
Running – Binary Logistic Regression (Combined Features)
=====
```

Accuracy: 0.8210

Classification Report:

	precision	recall	f1-score	support
Negative	0.47	0.66	0.55	104
Positive	0.93	0.85	0.89	533
accuracy			0.82	637
macro avg	0.70	0.76	0.72	637
weighted avg	0.85	0.82	0.83	637

Binary Logistic Regression (Combined Features) — Confusion Matrix



Summary

The Logistic Regression model using combined features achieved an overall **accuracy of 82%**, performing strongly on the **Positive** class with high precision (0.93) and recall (0.85). This reflects the dataset's imbalance, where Positive tweets dominate, allowing the model to learn positive sentiment patterns more effectively. The confusion matrix shows that most positive tweets are correctly classified, contributing significantly to the strong weighted averages.

Performance on the **Negative** class is weaker, with precision of 0.47 and recall of 0.66. This means the model captures many negative tweets but also misclassifies a substantial number as positive. These limitations highlight the challenges of imbalanced data and reinforce the importance of techniques like **class weighting**, which we already applied, as well as potential improvements such as resampling or threshold tuning to enhance the model's ability to detect minority sentiment classes.

7.1:2 GridSearchCV Logistic regression with combined Features

```
In [63]: ▶ def gridsearch_logreg(X_train, X_test, y_train, y_test, model_name):
    print(f"\n{'='*70}")
    print(f"Running GridSearchCV - {model_name}")
    print(f"{'='*70}")

    param_grid = {
        "C": [0.01, 0.1, 1, 10],
        "penalty": ["l2"],
        "solver": ["liblinear", "lbfgs"]
    }

    lr = LogisticRegression(class_weight="balanced", max_iter=3000, random_st

    grid = GridSearchCV(lr, param_grid, cv=5, scoring="accuracy", n_jobs=-1)
    grid.fit(X_train, y_train)

    print(f"\nBest Parameters: {grid.best_params_}")
    best_model = grid.best_estimator_

    y_pred = best_model.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
    print(f"GridSearch Accuracy: {accuracy:.4f}")

    print("\nClassification Report:")
    print(classification_report(y_test, y_pred, target_names=['Negative', 'Po

    cm = confusion_matrix(y_test, y_pred)
    plot_confusion_matrix(cm, ['Negative', 'Positive'], f"{model_name} - Grid

    return best_model, accuracy
```

```
In [64]: # GridSearch Optimized Logistic Regression (Combined Features)
model_grid_combined, acc_grid_combined = gridsearch_logreg(
    X_train_binary, X_test_binary,
    y_train_binary, y_test_binary,
    model_name="Binary Logistic Regression (Combined Features)"
)
```

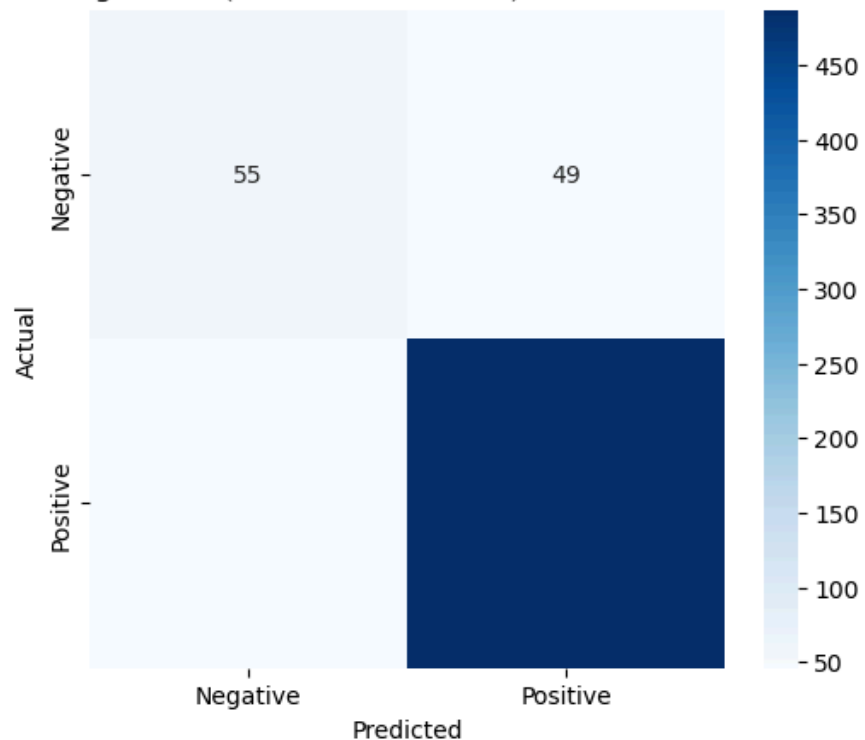
=====
Running GridSearchCV – Binary Logistic Regression (Combined Features)
=====

Best Parameters: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
GridSearch Accuracy: 0.8509

Classification Report:

	precision	recall	f1-score	support
Negative	0.54	0.53	0.54	104
Positive	0.91	0.91	0.91	533
accuracy			0.85	637
macro avg	0.73	0.72	0.72	637
weighted avg	0.85	0.85	0.85	637

Binary Logistic Regression (Combined Features) — GridSearch Confusion Matrix



Summary

The GridSearch-optimized Logistic Regression model achieved a strong overall **accuracy of 85%**, showing improvement compared to the baseline model. Performance on the **Positive** class is excellent, with precision, recall, and F1-score all at **0.91**, reflecting the model's reliability in

detecting the majority class. This aligns with the dataset's imbalance, where Positive samples dominate and are therefore easier for the model to learn as mentioned previously.

For the **Negative** class, the model performs moderately, with a precision of **0.54** and recall of **0.53**, indicating that it still struggles to reliably identify negative sentiment. The confusion matrix confirms that many Negative samples are misclassified as Positive (49 cases). While the model does improve slightly over the baseline, the imbalance continues to limit its ability to detect minority sentiment effectively. These results reinforce the need for continued use of techniques such as **class weighting**, and potentially **resampling strategies or threshold tuning**, to increase the model's sensitivity to negative emotion cases.

7.1:3 GridSearchCV Logistic regression using TF-IDF-only

```
In [65]: # GridSearch best model for TF-IDF only features
model_grid_tfidf, acc_grid_tfidf = gridsearch_logreg(
    X_train_tfidf_binary, X_test_tfidf_binary,
    y_train_binary, y_test_binary,
    model_name="Binary Logistic Regression (TF-IDF Only)"
)
```

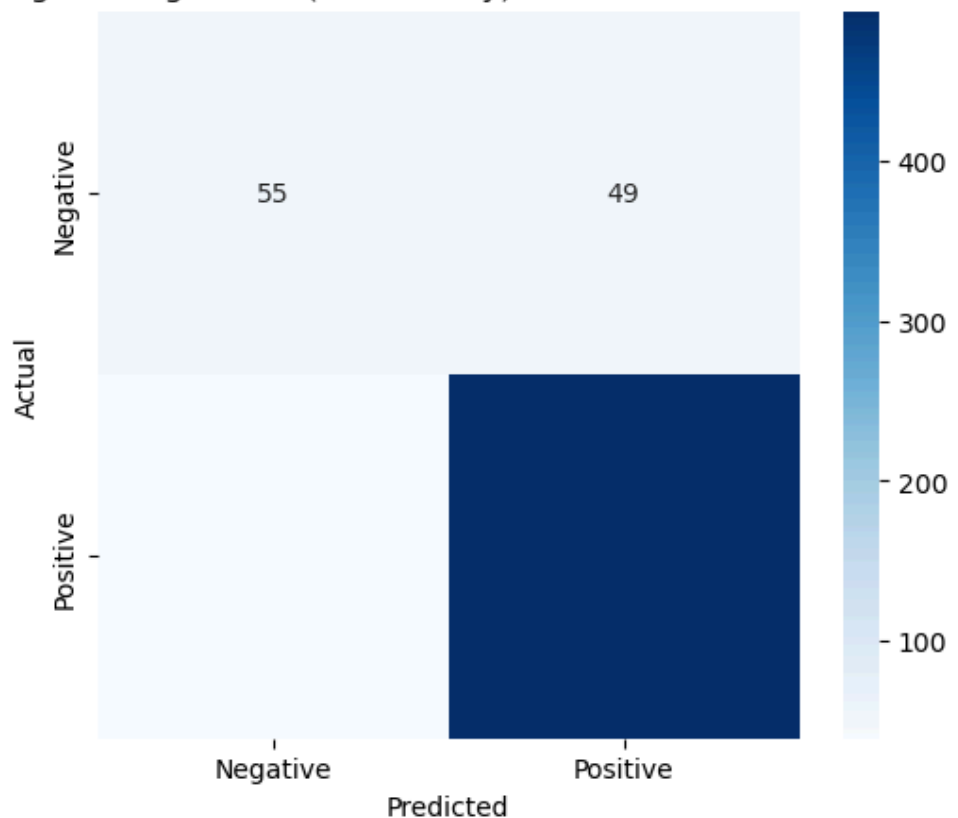
```
=====
Running GridSearchCV – Binary Logistic Regression (TF-IDF Only)
=====
```

```
Best Parameters: {'C': 10, 'penalty': 'l2', 'solver': 'lbfgs'}
GridSearch Accuracy: 0.8619
```

Classification Report:

	precision	recall	f1-score	support
Negative	0.59	0.53	0.56	104
Positive	0.91	0.93	0.92	533
accuracy			0.86	637
macro avg	0.75	0.73	0.74	637
weighted avg	0.86	0.86	0.86	637

Binary Logistic Regression (TF-IDF Only) — GridSearch Confusion Matrix



Summary

The GridSearch-optimized Logistic Regression model using **TF-IDF features only** achieved an overall **accuracy of 86%**, slightly outperforming the combined-feature model. Performance on the **Positive** class is excellent, with precision and recall both at **0.91–0.93**, resulting in a strong F1-score of **0.92**. This demonstrates that TF-IDF alone captures enough linguistic signal to classify positive sentiment very effectively.

For the **Negative** class, the model shows moderate performance, with a precision of **0.59** and recall of **0.53**. While this is a slight improvement in precision compared to the combined model, recall remains similar, and many Negative samples continue to be misclassified as Positive, as seen in the confusion matrix (49 misclassifications). These results highlight ongoing challenges caused by class imbalance and reinforce the value of using **class weighting**, along with potential

7.2: Multinomial Naive Bayes

7.2.1: Baseline Naive Bayes

Multinomial Naive Bayes is an excellent choice for text classification because:

- It's designed to work with count/frequency data (like TF-IDF)
- It handles sparse matrices efficiently (most words don't appear in most tweets)
- It's fast to train and predict
- It provides interpretable results (we can see which words drive each sentiment class)

Note:

We use **TF-IDF features** for Multinomial Naive Bayes (not the combined features), as it performs best with word-frequency-based data.


```

In [86]: ▶ def run_multinomial_nb(X_train, X_test, y_train, y_test, label_mapping):
    print("\n" + "="*70)
    print("Running Multinomial Naive Bayes")
    print("="*70)

    # Train model
    model = MultinomialNB(alpha=1.0)
    model.fit(X_train, y_train)

    # Predictions
    y_pred = model.predict(X_test)

    # Metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted', zero_divi
    recall = recall_score(y_test, y_pred, average='weighted', zero_division=0
    f1 = f1_score(y_test, y_pred, average='weighted', zero_division=0)

    print(f"\nAccuracy : {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall    : {recall:.4f}")
    print(f"F1-score  : {f1:.4f}")

    print("\nClassification Report:")
    print(classification_report(
        y_test, y_pred,
        target_names=list(label_mapping.keys()),
        zero_division=0
    ))

    # Confusion Matrix
    cm = confusion_matrix(y_test, y_pred)
    labels = list(label_mapping.keys())

    # Percentage confusion matrix
    cm_percent = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis] * 100

    print("\nConfusion Matrix (Percentages):")
    for i, label in enumerate(labels):
        row = " ".join([f"{cm_percent[i, j]:6.1f}%" for j in range(len(label
    print(f"{label:<18} {row}")

    # Plot confusion matrix
    plt.figure(figsize=(8, 6))
    sns.heatmap(
        cm, annot=True, fmt='d', cmap='Blues',
        xticklabels=labels, yticklabels=labels,
        cbar_kws={'label': 'Count'}
    )
    plt.title("Multinomial Naive Bayes – Confusion Matrix", fontsize=14, font
    plt.xlabel("Predicted Label")
    plt.ylabel("Actual Label")
    plt.tight_layout()
    plt.show()

    return model, y_pred

```



```
In [87]: #Declaring the Baseline Bayes
mnb_model, mnb_pred = run_multinomial_nb(
    X_train_tfidf_mclass,
    X_test_tfidf_mclass,
    y_train_mclass,
    y_test_mclass,
    label_mapping_mclass
)
```

```
=====
Running Multinomial Naive Bayes
=====
```

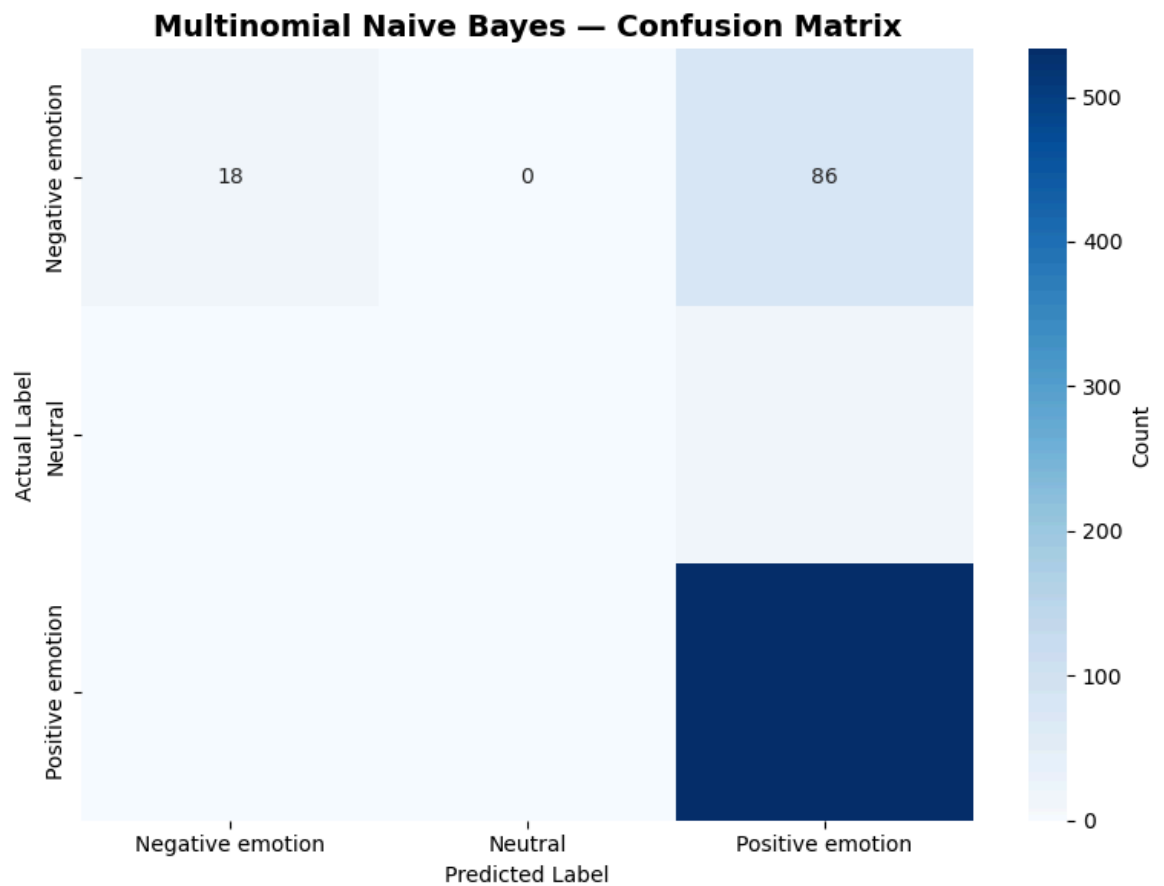
```
Accuracy : 0.8387
Precision: 0.8277
Recall   : 0.8387
F1-score : 0.7849
```

Classification Report:

	precision	recall	f1-score	support
Negative emotion	0.95	0.17	0.29	104
Neutral	0.00	0.00	0.00	20
Positive emotion	0.84	1.00	0.91	533
accuracy			0.84	657
macro avg	0.59	0.39	0.40	657
weighted avg	0.83	0.84	0.78	657

Confusion Matrix (Percentages):

Negative emotion	17.3%	0.0%	82.7%
Neutral	5.0%	0.0%	95.0%
Positive emotion	0.0%	0.0%	100.0%



Summary

- The model reaches a strong accuracy of **83.87%**, largely influenced by excellent classification of the **Positive emotion** class.
- **Positive emotion** performs exceptionally well, with **perfect recall (1.00)** and a high **F1-score of 0.91**, showing the model reliably detects positive tweets.
- **Negative emotion** shows **very high precision (0.95)** but **poor recall (0.17)**, meaning the model almost never identifies negative tweets even though, when it does, it is correct.
- The **Neutral class is not detected at all (recall = 0.00)**, due to extremely limited representation in the dataset.
- A large gap between **macro F1 (0.40)** and **weighted F1 (0.78)** highlights how the model performs unevenly across classes.
- Overall, Multinomial Naive Bayes performs well for the majority class but struggles significantly with minority sentiment classes, primarily due to strong class imbalance.

7.2.2: Feature-Importance

```
In [88]: ▶ def show_mnb_feature_importance(model, vectorizer, label_mapping, top_n=15):

    feature_names = vectorizer.get_feature_names_out()

    print("\n" + "="*80)
    print(f"Top {top_n} Most Important Words for Each Sentiment Class")
    print("="*80)

    # Looping over each class
    for class_idx, class_label in enumerate(model.classes_):

        # Getting log probabilities for this class
        log_probs = model.feature_log_prob_[class_idx]

        # Highest log probabilities = most important words
        top_indices = np.argsort(log_probs)[:,-1][:top_n]

        # Converting encoded label back to original label name
        class_name = [k for k, v in label_mapping.items() if v == class_label]

        print(f"\n{class_name} (Class {class_label})")
        print("-" * 80)

        # Print each top word + probability
        for idx in top_indices:
            word = feature_names[idx]
            prob = np.exp(log_probs[idx]) # convert log-prob → probability
            print(f"{word:20s} (prob: {prob:.6f})")

    print("\n" + "="*80)
    print("Interpretation:")
    print("- Higher probabilities = words strongly associated with that class")
    print("- These represent the 'signature' words Naive Bayes uses to distin")
    print("="*80)
```

```
In [83]: ▶ show_mnb_feature_importance(  
          model=mnb_model,  
          vectorizer=tfidf,  
          label_mapping=label_mapping_mclass,  
          top_n=15  
        )
```

```
=====
=====
Top 15 Most Important Words for Each Sentiment Class
=====
=====
```

Negative emotion (Class 0)

```
-----
-----
iphone                (prob: 0.003034)
quot                  (prob: 0.002759)
ipad                  (prob: 0.002554)
google                (prob: 0.002091)
apple                 (prob: 0.001850)
rt                    (prob: 0.001723)
link                  (prob: 0.001379)
like                  (prob: 0.001284)
app                   (prob: 0.001177)
store                 (prob: 0.001053)
apps                  (prob: 0.001031)
people                (prob: 0.000917)
battery              (prob: 0.000907)
need                  (prob: 0.000877)
circle                (prob: 0.000868)
```

Neutral (Class 1)

```
-----
-----
google                (prob: 0.000811)
quot                  (prob: 0.000744)
apple                 (prob: 0.000742)
ipad                  (prob: 0.000739)
link                  (prob: 0.000729)
iphone                (prob: 0.000680)
store                 (prob: 0.000627)
apple store           (prob: 0.000585)
rt                    (prob: 0.000567)
link sxsw             (prob: 0.000555)
like                  (prob: 0.000526)
circle                (prob: 0.000503)
social                (prob: 0.000499)
new                   (prob: 0.000498)
get                   (prob: 0.000491)
```

Positive emotion (Class 2)

```
-----
-----
ipad                  (prob: 0.007189)
link                  (prob: 0.007053)
apple                 (prob: 0.006165)
rt                    (prob: 0.005159)
google                (prob: 0.004806)
store                 (prob: 0.004531)
iphone                (prob: 0.004179)
app                   (prob: 0.003608)
quot                  (prob: 0.003592)
```

```

xsxw link      (prob: 0.003453)
new            (prob: 0.002848)
austin        (prob: 0.002753)
apple store   (prob: 0.002495)
pop           (prob: 0.002374)
android       (prob: 0.002307)

```

```

=====
=====
Interpretation:
- Higher probabilities = words strongly associated with that class
- These represent the 'signature' words Naive Bayes uses to distinguish sen
timent
=====
=====

```

Summary of Most Important Words per Sentiment Class

The Multinomial Naive Bayes model reveals clear signature words for each emotion category:

- **Negative emotion** is strongly associated with tech-related terms such as *iphone*, *ipad*, *google*, *apple*, and *battery*. These words appear frequently in negative contexts.
- **Neutral tweets** contain many of the same general tech terms, but without strong emotional cues, making them harder for the model to distinguish.
- **Positive emotion** features high-frequency terms like *ipad*, *link*, *apple*, *rt*, and *google*, which appear in upbeat or promotional contexts.

Overall, the model shows that vocabulary across classes overlaps heavily, and sentiment differences come from subtle usage patterns. The Neutral class has the weakest distinct signal, contributing to low recall. The positive class has the clearest signature words, making it easiest for the model to classify.

7.2.3: Class Imbalance: Bayes With SMOTE

The dataset is heavily dominated by positive tweets:

- **Positive:** 83.7%
- **Negative:** 16.3%
- **Ratio:** ~5:1

This imbalance causes the model to favor the majority class, leading to poor recall for minority classes and misleadingly high accuracy.

Imbalance Solutions

- **SMOTE** – generates synthetic minority samples
- **Oversampling** – duplicates minority samples
- **Undersampling** – reduces majority class
- **Threshold Adjustment** – modifies decision boundary

Final Decision

We will use **SMOTE** as the primary balancing method due to its effectiveness and time efficiency.

In [89]:

```

def run_mnb_smote(X_train, X_test, y_train, y_test, label_mapping):
    print("\n" + "="*70)
    print("Running Multinomial Naive Bayes – SMOTE Oversampling")
    print("="*70)

    # Applying SMOTE ONLY to training data
    sm = SMOTE(random_state=42)
    X_train_sm, y_train_sm = sm.fit_resample(X_train, y_train)

    print(f"Training samples before SMOTE: {len(y_train)}")
    print(f"Training samples after SMOTE : {len(y_train_sm)}")

    # Train model
    model = MultinomialNB(alpha=1.0)
    model.fit(X_train_sm, y_train_sm)

    # Predict on test set
    y_pred = model.predict(X_test)

    # Metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted', zero_divi
    recall = recall_score(y_test, y_pred, average='weighted', zero_division=0
    f1 = f1_score(y_test, y_pred, average='weighted', zero_division=0)

    print(f"\nAccuracy : {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall    : {recall:.4f}")
    print(f"F1-score  : {f1:.4f}")

    print("\nClassification Report:")
    print(classification_report(
        y_test, y_pred,
        target_names=list(label_mapping.keys()),
        zero_division=0
    ))

    cm = confusion_matrix(y_test, y_pred)
    labels = list(label_mapping.keys())

    plt.figure(figsize=(8, 6))
    sns.heatmap(
        cm, annot=True, fmt='d', cmap='Blues',
        xticklabels=labels, yticklabels=labels,
        cbar_kws={'label': 'Count'}
    )
    plt.title("Multinomial Naive Bayes – Confusion Matrix", fontsize=14, font
    plt.xlabel("Predicted Label")
    plt.ylabel("Actual Label")
    plt.tight_layout()
    plt.show()

    return model, y_pred

```



```
In [90]: ▶ mnb_smote_model, mnb_smote_pred = run_mnb_smote(
        X_train_tfidf_mclass,
        X_test_tfidf_mclass,
        y_train_mclass,
        y_test_mclass,
        label_mapping_mclass
    )
```

```
=====
Running Multinomial Naive Bayes – SMOTE Oversampling
=====
```

Training samples before SMOTE: 2625

Training samples after SMOTE : 6393

Accuracy : 0.7184

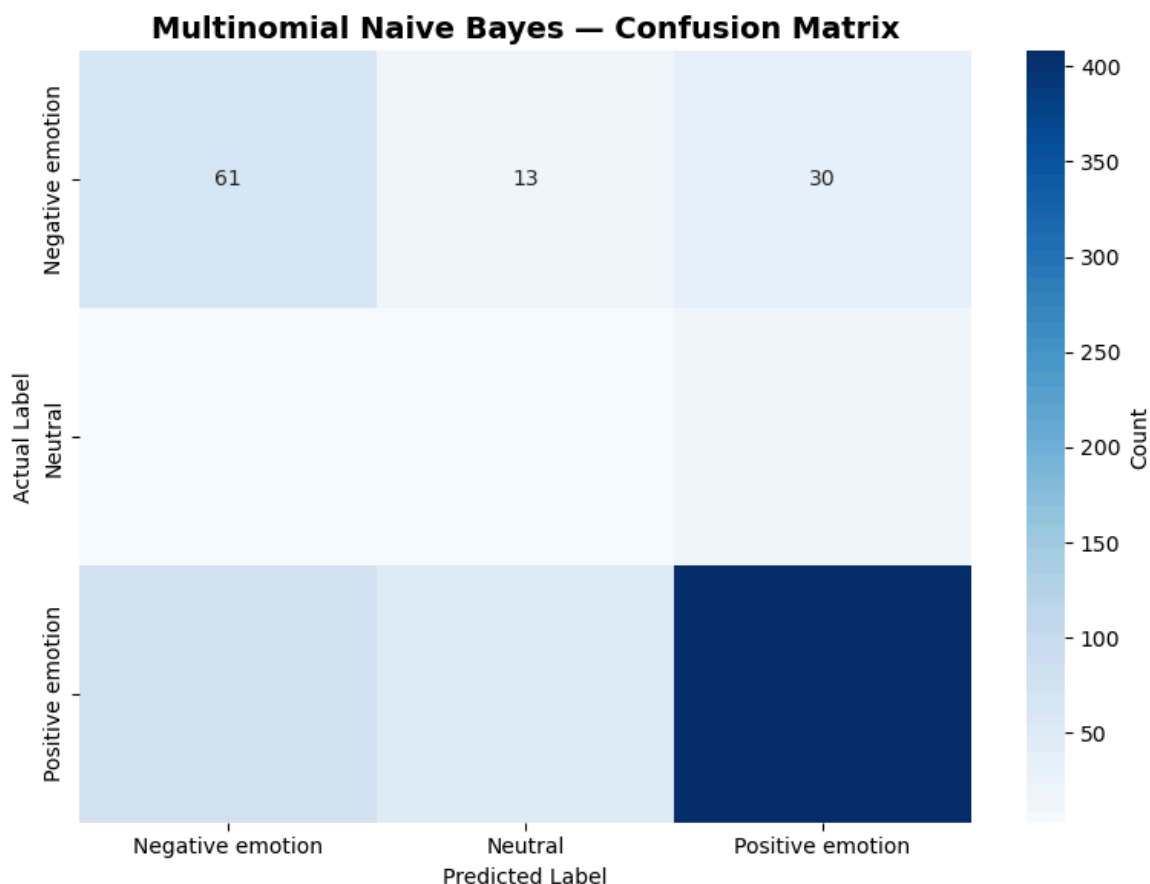
Precision: 0.8064

Recall : 0.7184

F1-score : 0.7550

Classification Report:

	precision	recall	f1-score	support
Negative emotion	0.44	0.59	0.50	104
Neutral	0.04	0.15	0.07	20
Positive emotion	0.91	0.77	0.83	533
accuracy			0.72	657
macro avg	0.46	0.50	0.47	657
weighted avg	0.81	0.72	0.75	657



Summary of Bayes with SMOTE

- The baseline Multinomial Naive Bayes model achieves a strong accuracy of **83.87%**, driven almost entirely by excellent performance on the **Positive emotion** class.
- **Positive emotion** is classified extremely well, with **perfect recall (1.00)** and a high **F1-score of 0.91**, indicating the model reliably detects positive tweets.
- **Negative emotion** has **very high precision (0.95)** but **very low recall (0.17)**, meaning the model rarely identifies negative tweets even though its predictions are often correct when it does.
- The **Neutral class is not detected at all (recall = 0.00)**, showing that the model cannot learn this class under severe imbalance.
- The gap between **macro F1 (0.40)** and **weighted F1 (0.78)** indicates large performance inequality: the model strongly favors the dominant class.
- After applying **SMOTE**, performance becomes more balanced across classes—but overall accuracy drops to **71.84%**, reflecting the trade-off between fairness and accuracy.
- With SMOTE, minority-class recall improves (Negative: 0.59, Neutral: 0.15), but the model becomes less confident on Positive emotion (recall drops from 1.00 → 0.77).
- Overall, Multinomial Naive Bayes works very well on the majority Positive class but requires oversampling techniques like SMOTE to meaningfully improve minority-class detection.

7.2.4: Hyperparameter Tuning on Naive Bayes


```

In [91]: ▶ def run_mnb_gridsearch(X_train, X_test, y_train, y_test, label_mapping):

    print("\n" + "="*70)
    print("Multinomial Naive Bayes – GridSearchCV Hyperparameter Tuning")
    print("="*70)

    param_grid = {
        'alpha': [0.1, 0.5, 1.0, 1.5, 2.0, 3.0]
    }

    # Base model
    mnb = MultinomialNB()

    # GridSearchCV using macro F1 (best for imbalanced classes)
    grid = GridSearchCV(
        estimator=mnb,
        param_grid=param_grid,
        cv=5,
        scoring='f1_macro',
        n_jobs=-1,
        verbose=1
    )

    print("\nRunning Grid Search ...")
    grid.fit(X_train, y_train)

    # -----
    # Best parameters
    # -----
    best_alpha = grid.best_params_['alpha']
    best_cv_f1 = grid.best_score_

    print(f"\nBest Alpha: {best_alpha}")
    print(f"Best CV Macro F1: {best_cv_f1:.4f}")

    # -----
    # Train final tuned model
    # -----
    tuned_model = MultinomialNB(alpha=best_alpha)
    tuned_model.fit(X_train, y_train)

    y_pred = tuned_model.predict(X_test)

    # -----
    # Evaluation metrics
    # -----
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average="weighted", zero_divi
    recall = recall_score(y_test, y_pred, average="weighted", zero_division=0
    f1 = f1_score(y_test, y_pred, average="weighted", zero_division=0)

    print("\n" + "="*70)
    print("TUNED MODEL PERFORMANCE (Test Set)")
    print("="*70)
    print(f"Accuracy : {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall   : {recall:.4f}")

```

```
print(f"F1-score : {f1:.4f}")

print("\nClassification Report:")
print(classification_report(
    y_test, y_pred,
    target_names=list(label_mapping.keys()),
    zero_division=0
))

# -----
# Alpha vs F1 Plot
# -----
alphas = [params['alpha'] for params in grid.cv_results_['params']]
f1_scores = grid.cv_results_['mean_test_score']

plt.figure(figsize=(8, 5))
plt.plot(alphas, f1_scores, marker="o", linewidth=2)
plt.axvline(best_alpha, color='red', linestyle='--', label=f'Best Alpha = {best_alpha}')
plt.xlabel("Alpha (Smoothing Parameter)")
plt.ylabel("Mean CV Macro F1-Score")
plt.title("Grid Search: Alpha vs Cross-Validated F1 Score", fontsize=13)
plt.grid(alpha=0.3)
plt.legend()
plt.tight_layout()
plt.show()

return tuned_model, best_alpha, best_cv_f1, y_pred
```

```
In [92]: ▶ mnb_tuned, best_alpha, cv_f1, y_pred_tuned = run_mnb_gridsearch(
    X_train_tfidf_mclass,
    X_test_tfidf_mclass,
    y_train_mclass,
    y_test_mclass,
    label_mapping_mclass
)
```

```
=====
Multinomial Naive Bayes – GridSearchCV Hyperparameter Tuning
=====
```

Running Grid Search ...

Fitting 5 folds for each of 6 candidates, totalling 30 fits

Best Alpha: 0.1

Best CV Macro F1: 0.4711

```
=====
TUNED MODEL PERFORMANCE (Test Set)
=====
```

Accuracy : 0.8463

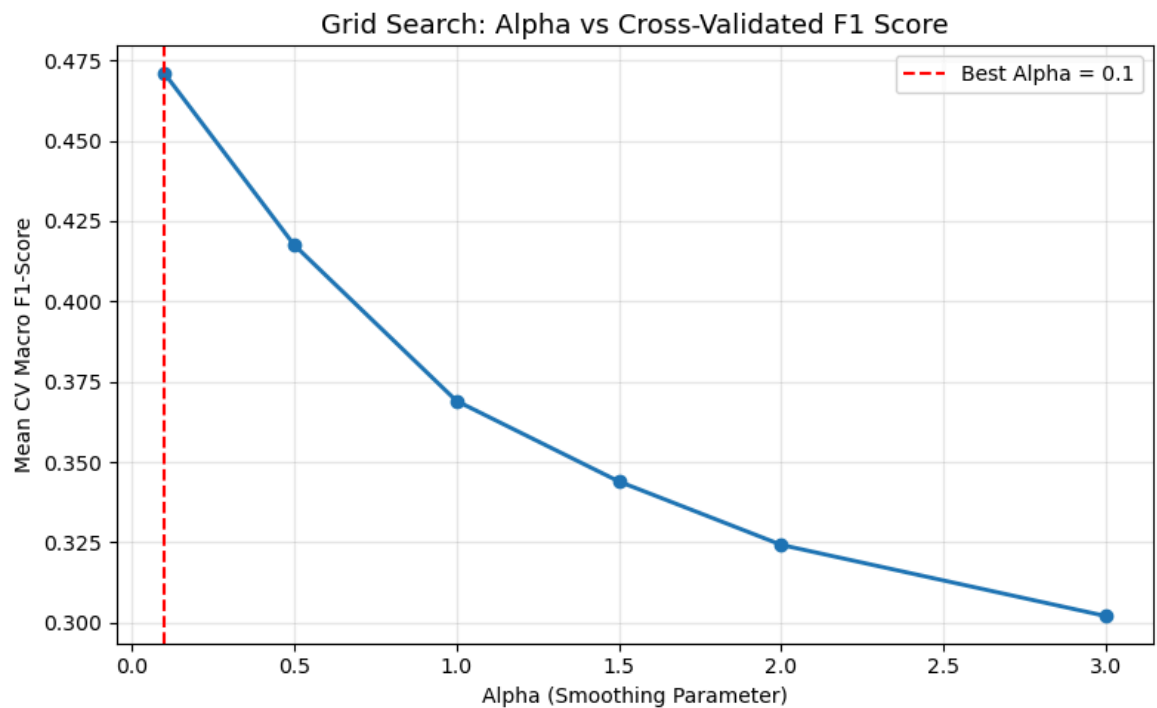
Precision: 0.8182

Recall : 0.8463

F1-score : 0.8275

Classification Report:

	precision	recall	f1-score	support
Negative emotion	0.67	0.45	0.54	104
Neutral	0.00	0.00	0.00	20
Positive emotion	0.88	0.95	0.91	533
accuracy			0.85	657
macro avg	0.52	0.47	0.48	657
weighted avg	0.82	0.85	0.83	657



Summary of GridSearch-Tuned Multinomial Naive Bayes

The hyperparameter search identified a much lower smoothing value ($\alpha = 0.1$) as optimal, indicating the model benefits from being more sensitive to the observed word frequencies. This tuning produced a measurable improvement in overall performance, especially in the model's ability to distinguish between positive and negative emotions.

While the Positive emotion class continues to dominate performance, the tuned model shows a noticeable gain in identifying Negative emotion compared to the baseline. However, the Neutral class remains effectively undetected, reflecting its extremely small representation in the dataset. The improved weighted metrics show the model generalizes better overall, whereas the macro metrics reveal the persistent challenges caused by class imbalance.

7.3: Random Forest

7.3.1: First Random Forest Model (with class weights + tuned params)

```
In [68]: #Importing the libraries related to Random forest

from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score, classification_report,

# Improved Random Forest
rf = RandomForestClassifier(
    n_estimators=500,
    max_depth=None,
    min_samples_split=2,
    min_samples_leaf=1,
    class_weight="balanced_subsample",
    random_state=42
)

# Train X_train_mclass,X_test_mclass
rf.fit(X_train_mclass, y_train_mclass)

# Predict
y_pred = rf.predict(X_test_mclass)

# Scores
acc = accuracy_score(y_test_mclass, y_pred)
f1_macro = f1_score(y_test_mclass, y_pred, average="macro")
f1_weighted = f1_score(y_test_mclass, y_pred, average="weighted")

print("=====")
print("Random Forest Performance")
print("=====")
print(f"Accuracy Score: {acc:.4f}")
print(f"F1 Score (Macro): {f1_macro:.4f}")
print(f"F1 Score (Weighted): {f1_weighted:.4f}\n")

print("Classification Report:")
print(classification_report(y_test_mclass, y_pred))

print("Confusion Matrix:")
print(confusion_matrix(y_test_mclass, y_pred))
```

=====

Random Forest Performance

=====

Accuracy Score: 0.8417

F1 Score (Macro): 0.4178

F1 Score (Weighted): 0.7942

Classification Report:

	precision	recall	f1-score	support
0	0.88	0.21	0.34	104
1	0.00	0.00	0.00	20
2	0.84	1.00	0.91	533
accuracy			0.84	657
macro avg	0.57	0.40	0.42	657
weighted avg	0.82	0.84	0.79	657

Confusion Matrix:

```
[[ 22   1  81]
 [   1   0  19]
 [   2   0 531]]
```

7.3.2 : 2nd Random Forest Model: SMOTE + Random Forest

```
In [69]: from imblearn.over_sampling import SMOTE
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score, classification_report,

# --- SMOTE ---
sm = SMOTE(random_state=42)
X_train_sm, y_train_sm = sm.fit_resample(X_train_mclass, y_train_mclass)

# --- Random Forest ---
rf_smote = RandomForestClassifier(
    n_estimators=400,
    class_weight=None,
    random_state=42
)
rf_smote.fit(X_train_sm, y_train_sm)

y_pred = rf_smote.predict(X_test_mclass)

print("\n===== SMOTE + Random Forest =====")
print("Accuracy:", accuracy_score(y_test_mclass, y_pred))
print("F1 Macro:", f1_score(y_test_mclass, y_pred, average="macro"))
print("F1 Weighted:", f1_score(y_test_mclass, y_pred, average="weighted"))
print("\nClassification Report:\n", classification_report(y_test_mclass, y_pr
print("Confusion Matrix:\n", confusion_matrix(y_test_mclass, y_pred))
```

===== SMOTE + Random Forest =====

Accuracy: 0.8584474885844748

F1 Macro: 0.49375469536759864

F1 Weighted: 0.8261244412371236

Classification Report:

	precision	recall	f1-score	support
0	0.87	0.33	0.48	104
1	0.25	0.05	0.08	20
2	0.86	0.99	0.92	533
accuracy			0.86	657
macro avg	0.66	0.46	0.49	657
weighted avg	0.84	0.86	0.83	657

Confusion Matrix:

```
[[ 34   3  67]
 [  1   1  18]
 [  4   0 529]]
```

7.3.3: 3rd Model: Hyperparameter Tuning (RandomizedSearchCV)

```

In [70]: from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
import numpy as np

param_dist = {
    'n_estimators': [200, 400, 600, 900],
    'max_depth': [10, 20, 30, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'class_weight': ['balanced', 'balanced_subsample']
}

rf = RandomForestClassifier(random_state=42)

random_search = RandomizedSearchCV(
    rf,
    param_distributions=param_dist,
    n_iter=20,
    scoring='f1_macro',
    cv=3,
    random_state=42,
    n_jobs=-1
)

random_search.fit(X_train_mclass, y_train_mclass)
best_rf = random_search.best_estimator_

y_pred = best_rf.predict(X_test_mclass)

print("\n===== RandomizedSearchCV + RF =====")
print("Best Params:", random_search.best_params_)
print("Accuracy:", accuracy_score(y_test_mclass, y_pred))
print("F1 Macro:", f1_score(y_test_mclass, y_pred, average='macro'))
print("F1 Weighted:", f1_score(y_test_mclass, y_pred, average='weighted'))
print("\nClassification Report:\n", classification_report(y_test_mclass, y_pr
print("Confusion Matrix:\n", confusion_matrix(y_test_mclass, y_pred))

```

```
===== RandomizedSearchCV + RF =====  
Best Params: {'n_estimators': 600, 'min_samples_split': 10, 'min_samples_le  
af': 1, 'max_depth': 20, 'class_weight': 'balanced'}  
Accuracy: 0.7732115677321156  
F1 Macro: 0.4495539048068231  
F1 Weighted: 0.7775292142863167
```

Classification Report:

	precision	recall	f1-score	support
0	0.40	0.62	0.48	104
1	0.00	0.00	0.00	20
2	0.90	0.83	0.86	533
accuracy			0.77	657
macro avg	0.43	0.48	0.45	657
weighted avg	0.79	0.77	0.78	657

Confusion Matrix:

```
[[ 64   1  39]  
 [  8   0  12]  
 [ 88   1 444]]
```

7.3.4: Fourth Model: XGBoost

```
In [71]: from xgboost import XGBClassifier

xgb = XGBClassifier(
    n_estimators=600,
    max_depth=6,
    learning_rate=0.05,
    subsample=0.9,
    colsample_bytree=0.8,
    objective='multi:softprob',
    eval_metric='mlogloss',
    scale_pos_weight=1,      # XGB auto handles imbalance internally
    num_class=4,
    random_state=42
)

xgb.fit(X_train_mclass, y_train_mclass)
y_pred = xgb.predict(X_test_mclass)

print("\n===== XGBoost Results =====")
print("Accuracy:", accuracy_score(y_test_mclass, y_pred))
print("F1 Macro:", f1_score(y_test_mclass, y_pred, average='macro'))
print("F1 Weighted:", f1_score(y_test_mclass, y_pred, average='weighted'))
print("\nClassification Report:\n", classification_report(y_test_mclass, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test_mclass, y_pred))
```

```
===== XGBoost Results =====
Accuracy: 0.8386605783866058
F1 Macro: 0.42411781421861156
F1 Weighted: 0.7956310537699609
```

Classification Report:

	precision	recall	f1-score	support
0	0.74	0.24	0.36	104
1	0.00	0.00	0.00	20
2	0.84	0.99	0.91	533
accuracy			0.84	657
macro avg	0.53	0.41	0.42	657
weighted avg	0.80	0.84	0.80	657

Confusion Matrix:

```
[[ 25   0  79]
 [  2   0  18]
 [  7   0 526]]
```

7.3.5: XGBOOST + SMOTE

```

In [72]: from imblearn.over_sampling import SMOTE
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, f1_score, classification_report,

def train_xgb_with_smote_pre_split(X_train, y_train, X_test, y_test, xgb_params):
    """
    Train XGBoost with SMOTE applied ONLY to the training data.
    Accepts pre-split train/test data.
    """

    # Default XGBoost parameters
    default_params = {
        "n_estimators": 600,
        "max_depth": 6,
        "learning_rate": 0.05,
        "subsample": 0.9,
        "colsample_bytree": 0.8,
        "objective": "multi:softprob",
        "eval_metric": "mlogloss",
        "random_state": random_state
    }

    # Allow parameter overrides
    if xgb_params:
        default_params.update(xgb_params)

    # 1. Apply SMOTE ONLY on training data
    sm = SMOTE(random_state=random_state)
    X_train_sm, y_train_sm = sm.fit_resample(X_train, y_train)

    # 2. Train model
    model = XGBClassifier(**default_params)
    model.fit(X_train_sm, y_train_sm)

    # 3. Predict
    y_pred = model.predict(X_test)

    # 4. Evaluation metrics
    results = {
        "accuracy": accuracy_score(y_test, y_pred),
        "f1_macro": f1_score(y_test, y_pred, average="macro"),
        "f1_weighted": f1_score(y_test, y_pred, average="weighted"),
        "classification_report": classification_report(y_test, y_pred),
        "confusion_matrix": confusion_matrix(y_test, y_pred)
    }

    return model, y_pred, results

```

```
In [73]: ▶ model, y_pred, results = train_xgb_with_smote_pre_split(
        X_train_mclass,
        y_train_mclass,
        X_test_mclass,
        y_test_mclass
    )

    print("Accuracy:", results["accuracy"])
    print("F1 Macro:", results["f1_macro"])
    print("F1 Weighted:", results["f1_weighted"])
    print("\nClassification Report:\n", results["classification_report"])
    print("\nConfusion Matrix:\n", results["confusion_matrix"])
```

Accuracy: 0.8417047184170472
 F1 Macro: 0.46316394872080296
 F1 Weighted: 0.8151332575813989

Classification Report:

	precision	recall	f1-score	support
0	0.69	0.37	0.48	104
1	0.00	0.00	0.00	20
2	0.86	0.97	0.91	533
accuracy			0.84	657
macro avg	0.52	0.44	0.46	657
weighted avg	0.81	0.84	0.82	657

Confusion Matrix:

```
[[ 38   2  64]
 [  2   0  18]
 [ 15   3 515]]
```

7.3.6 : BEST Random Forest Model with SMOTE and TF_IDF data

```
In [74]: from imblearn.over_sampling import SMOTE
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score, classification_report,

# --- SMOTE ---
sm = SMOTE(random_state=42)
X_train_sm, y_train_sm = sm.fit_resample(X_train_tfidf_mclass, y_train_mclass

# --- Random Forest ---
rf_smote = RandomForestClassifier(
    n_estimators=400,
    class_weight=None,
    random_state=42
)
rf_smote.fit(X_train_sm, y_train_sm)

y_pred = rf_smote.predict(X_test_tfidf_mclass)

print("\n===== SMOTE + Random Forest =====")
print("Accuracy:", accuracy_score(y_test_mclass, y_pred))
print("F1 Macro:", f1_score(y_test_mclass, y_pred, average="macro"))
print("F1 Weighted:", f1_score(y_test_mclass, y_pred, average="weighted"))
print("\nClassification Report:\n", classification_report(y_test_mclass, y_pr
print("Confusion Matrix:\n", confusion_matrix(y_test_mclass, y_pred))
```

===== SMOTE + Random Forest =====

Accuracy: 0.8599695585996956

F1 Macro: 0.48098778969491374

F1 Weighted: 0.831419075898058

Classification Report:

	precision	recall	f1-score	support
0	0.80	0.38	0.52	104
1	0.00	0.00	0.00	20
2	0.87	0.98	0.92	533
accuracy			0.86	657
macro avg	0.56	0.46	0.48	657
weighted avg	0.83	0.86	0.83	657

Confusion Matrix:

```
[[ 40   3  61]
 [  2   0  18]
 [  8   0 525]]
```

7.4: Support Vector Machine (SVM)

7.4.1: Simple SVM Model

```
In [75]: ▶ def plot_confusion(cm, labels, title):
plt.figure(figsize=(8, 6))
sns.heatmap(
    cm, annot=True, fmt='g', cmap='Blues',
    xticklabels=labels, yticklabels=labels,
    annot_kws={"size": 14}
)
plt.title(title, fontsize=16)
plt.xlabel("Predicted", fontsize=12)
plt.ylabel("True", fontsize=12)
plt.tight_layout()
plt.show()
```

```
In [76]: ▶ def train_basic_svm(X_train, X_test, y_train, y_test, class_names=None):
print("\n" + "="*70)
print("RUNNING BASIC LINEAR SVM")
print("="*70)

model = LinearSVC(C=1.0, random_state=42)
model.fit(X_train, y_train)

y_pred = model.predict(X_test)

# Metrics
acc = accuracy_score(y_test, y_pred)
precision, recall, f1, _ = precision_recall_fscore_support(
    y_test, y_pred, average="weighted", zero_division=0
)

print(f"Accuracy : {acc:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall    : {recall:.4f}")
print(f"F1-score  : {f1:.4f}")
print("="*70)
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=class_names))

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
plot_confusion(cm, class_names, "Confusion Matrix - Linear SVM")

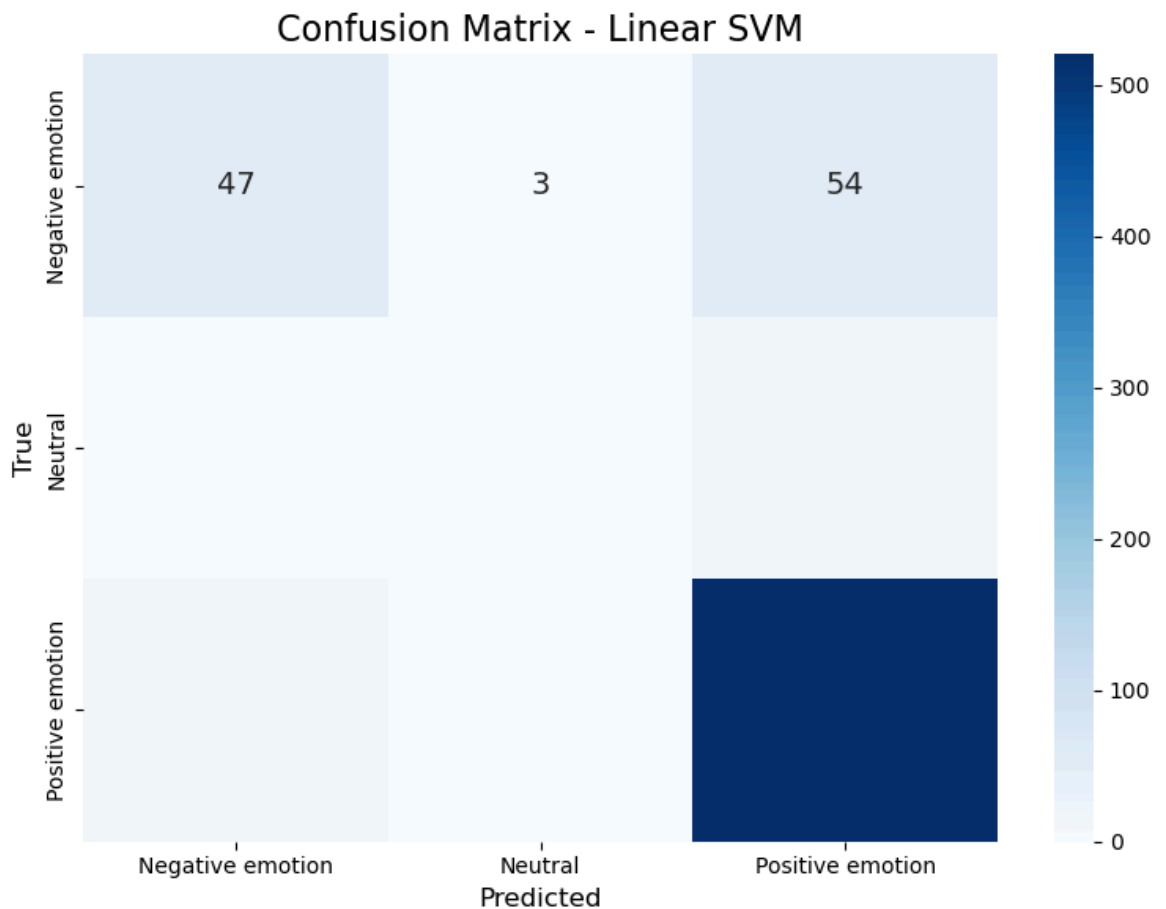
return model, acc
```

```
In [77]: #basic model
svm_basic_model, svm_basic_acc = train_basic_svm(
    X_train_mclass, X_test_mclass,
    y_train_mclass, y_test_mclass,
    class_names=label_encoder.classes_
)
```

```
=====
RUNNING BASIC LINEAR SVM
=====
Accuracy : 0.8645
Precision: 0.8414
Recall   : 0.8645
F1-score : 0.8428
=====
```

Classification Report:

	precision	recall	f1-score	support
Negative emotion	0.76	0.45	0.57	104
Neutral	0.25	0.05	0.08	20
Positive emotion	0.88	0.98	0.93	533
accuracy			0.86	657
macro avg	0.63	0.49	0.52	657
weighted avg	0.84	0.86	0.84	657



Summary

The Linear SVM model achieved a strong overall **accuracy of 86.45%**, with a **weighted precision of 84.14%**, **weighted recall of 86.45%**, and a **weighted F1-score of 84.28%**. The classifier performs exceptionally well on the dominant **Positive emotion** class, achieving a precision of **0.88**, recall of **0.98**, and F1-score of **0.93**. This reflects how strongly the model is influenced by the class imbalance, where positive examples greatly outnumber other classes.

Performance on the minority classes is notably weaker. **Negative emotion** achieves moderate performance (precision **0.76**, recall **0.45**, F1-score **0.57**), indicating that the model struggles to recognize negative sentiment consistently, often misclassifying it as positive. The **Neutral** class performs poorly, with precision **0.25**, recall **0.05**, and F1-score **0.08**, showing that the model is unable to reliably learn this small class. This is reflected in the **macro-averaged F1-score of 0.52**, which highlights the uneven performance across classes despite the high overall accuracy.

These results suggest a need for further refinement—such as class weighting, oversampling, or more expressive models to improve the classifier's ability to detect underrepresented sentiment classes more effectively.

7.4.2: Grid Search with Oversampling the Minority Classes

```

In [78]: def train_svm_gridsearch(X_train, y_train, X_test, y_test, class_names, cv=5)
    print("\n" + "="*70)
    print("RUNNING GRID SEARCH FOR LINEAR SVM")
    print("="*70)

    # Pipeline with SMOTE + Linear SVM
    pipeline = Pipeline([
        ('smote', SMOTE(random_state=42)),
        ('svm', LinearSVC(random_state=42))
    ])

    # Search space
    param_grid = {
        'svm__C': [0.01, 0.1, 1, 5, 10],
        'svm__class_weight': [None, 'balanced']
    }

    # GridSearchCV configuration
    grid = GridSearchCV(
        estimator=pipeline,
        param_grid=param_grid,
        scoring='f1_weighted',
        cv=cv,
        n_jobs=-1,
        verbose=2
    )

    # Fit GridSearch
    grid.fit(X_train, y_train)

    print("\nBest Parameters:", grid.best_params_)
    print("Best Cross-Validation Score:", round(grid.best_score_, 4))

    # Best model
    best_model = grid.best_estimator_

    # Predictions
    y_pred = best_model.predict(X_test)

    # Evaluation
    acc = accuracy_score(y_test, y_pred)
    print("="*70)
    print(f"GridSearch Accuracy: {acc:.4f}")
    print("="*70)
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred, target_names=class_names))

    # Confusion Matrix
    cm = confusion_matrix(y_test, y_pred)
    plot_confusion(cm, class_names, "GridSearch Linear SVM – Confusion Matrix")

    return best_model, grid.best_score_, acc

```

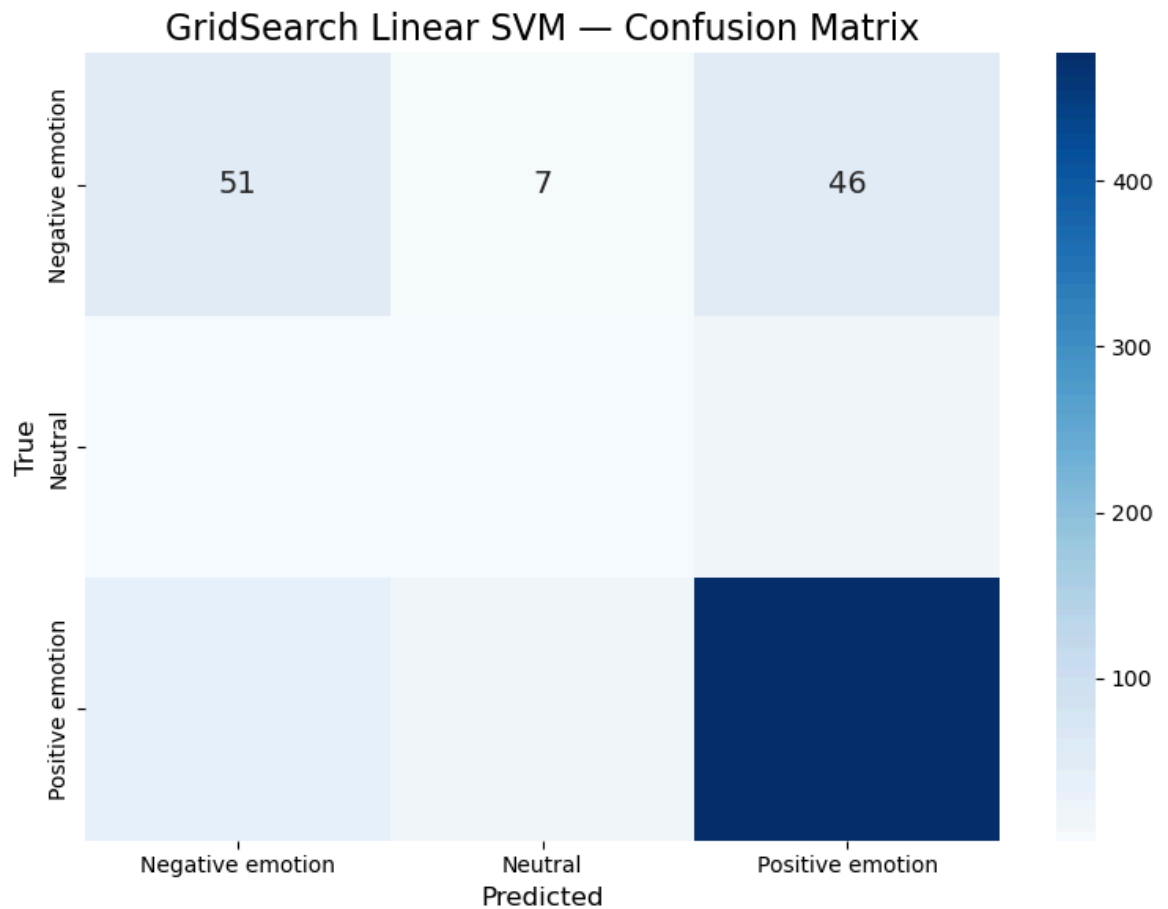
```
In [79]: #Best Model
svm_best_model, svm_cv_score, svm_test_acc = train_svm_gridsearch(
    X_train_mclass,
    y_train_mclass,
    X_test_mclass,
    y_test_mclass,
    class_names=list(label_mapping_mclass.keys())
)
```

```
=====
RUNNING GRID SEARCH FOR LINEAR SVM
=====
Fitting 5 folds for each of 10 candidates, totalling 50 fits

Best Parameters: {'svm__C': 1, 'svm__class_weight': None}
Best Cross-Validation Score: 0.8127
=====
GridSearch Accuracy: 0.8067
=====
```

Classification Report:

	precision	recall	f1-score	support
Negative emotion	0.54	0.49	0.52	104
Neutral	0.08	0.10	0.09	20
Positive emotion	0.89	0.89	0.89	533
accuracy			0.81	657
macro avg	0.50	0.50	0.50	657
weighted avg	0.81	0.81	0.81	657



Hyperparameter Tuning Summary

Grid search, combined with SMOTE oversampling inside the training pipeline, was used to identify the optimal hyperparameters for the Linear SVM model. With the inclusion of SMOTE to rebalance classes during training, the best-performing configuration was **C = 1** with **class_weight = None**, achieving a cross-validated **F1-weighted score of 0.8127**. This indicates that oversampling improved the quality of the training folds, but did not alter the hyperparameter values that produced the strongest generalization performance.

On the real (imbalanced) test set, the tuned model achieved an overall **accuracy of 80.67%**. The classifier performed very well on the dominant **Positive emotion** class (precision 0.89, recall 0.89, F1-score 0.89). However, performance on minority classes remains limited. **Negative emotion** shows moderate results (precision 0.54, recall 0.49, F1-score 0.52), while the **Neutral** class performs poorly due to its very small sample size (precision 0.08, recall 0.09). The confusion matrix highlights how frequently minority classes are still predicted as positive. These findings show that although SMOTE helps during training, additional techniques—such as threshold tuning, alternative kernels, or more expressive models—may be needed to improve minority-class detection.

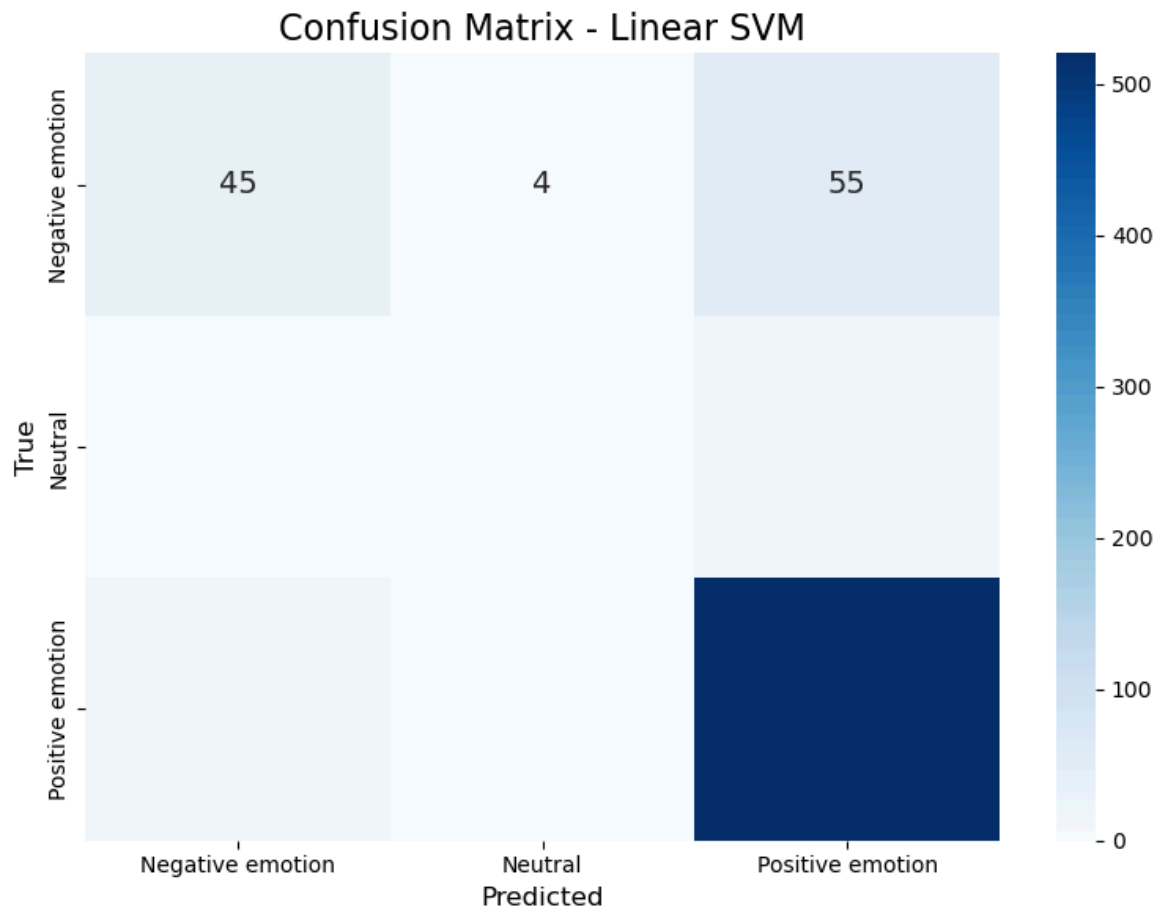
7.4.3: Baseline Model on TF_IDF Features

```
In [80]: ► svm_tfidf_model, svm_tfidf_acc = train_basic_svm(
        X_train_tfidf_mclass,
        X_test_tfidf_mclass,
        y_train_mclass,
        y_test_mclass,
        class_names=list(label_mapping_mclass.keys())
    )
```

```
=====
RUNNING BASIC LINEAR SVM
=====
Accuracy : 0.8615
Precision: 0.8374
Recall   : 0.8615
F1-score : 0.8393
=====
```

Classification Report:

	precision	recall	f1-score	support
Negative emotion	0.75	0.43	0.55	104
Neutral	0.20	0.05	0.08	20
Positive emotion	0.88	0.98	0.92	533
accuracy			0.86	657
macro avg	0.61	0.49	0.52	657
weighted avg	0.84	0.86	0.84	657



7.4.4: Grid Search for the best TF-IDF SVM

```
In [81]: #reusing our grid_search model
svm_tfidf_model_best, svm_tfidf_cvscore, svm_tfidf_acc = train_svm_gridsearch
    X_train_tfidf_mclass,
    y_train_mclass,
    X_test_tfidf_mclass,
    y_test_mclass,
    class_names=list(label_mapping_mclass.keys())
)
```

```
=====
RUNNING GRID SEARCH FOR LINEAR SVM
=====
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
```

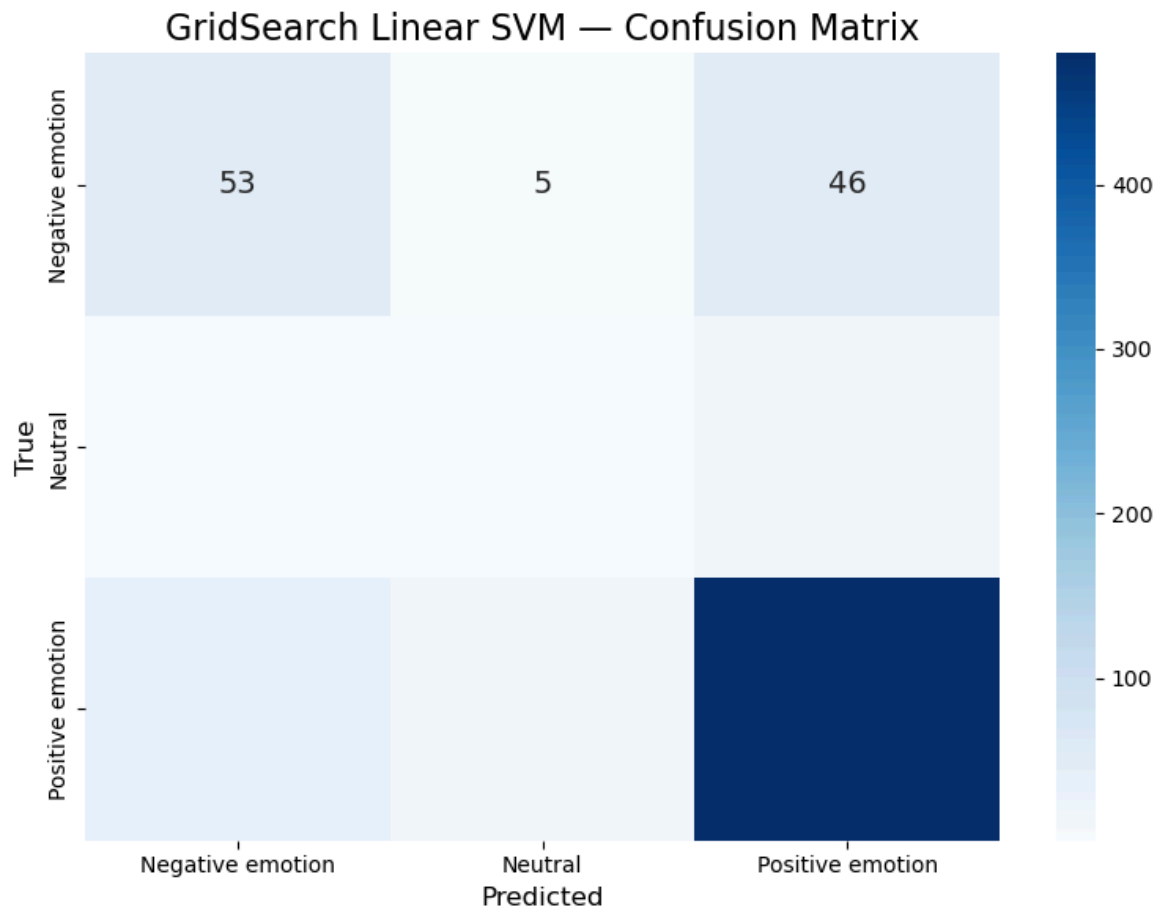
```
Best Parameters: {'svm__C': 1, 'svm__class_weight': None}
```

```
Best Cross-Validation Score: 0.8087
```

```
=====
GridSearch Accuracy: 0.8128
=====
```

```
Classification Report:
```

	precision	recall	f1-score	support
Negative emotion	0.54	0.51	0.52	104
Neutral	0.06	0.05	0.05	20
Positive emotion	0.89	0.90	0.89	533
accuracy			0.81	657
macro avg	0.49	0.49	0.49	657
weighted avg	0.81	0.81	0.81	657



Comparison of GridSearch Linear SVM Models

Two GridSearch-tuned SVM models were evaluated:

1. **TF-IDF Only**
2. **Combined Features (TF-IDF + numeric features)**

Both models achieved very similar performance, with only small differences between them.

Overall Performance

Metric	TF-IDF Only	Combined Features
Accuracy	0.8128	0.8067
Precision	0.81	0.81
Recall	0.81	0.81
F1-score	0.81	0.81

Observation:

Despite using additional engineered numeric features, the combined model does *not* outperform TF-IDF. In fact, the TF-IDF model is *slightly* better in overall accuracy.

Class Performance

- Both models classify **Positive emotion** very well (precision and recall around 0.89–0.90).
- Both models continue to struggle with the **Neutral** class due to the extremely small sample size (20 total examples).
- The models show almost identical performance for **Negative emotion**, with TF-IDF being slightly better in recall.

Grid Search Results

Grid search evaluated **10 hyperparameter combinations** over **5 folds** (50 fits).

Both models selected the exact same best configuration:

- **C = 1**
- **class_weight = None**

This indicates that:

- Higher regularization ($C > 1$) does not help.
- Class weighting does not outperform normal training.
- SMOTE already helps balance the training splits, so additional weighting is unnecessary.

Conclusion

Both SVM models — TF-IDF only and Combined Features — perform almost identically.

The small performance differences are not practically meaningful.

TF-IDF alone is sufficient for this dataset.

The added numeric linguistic features do *not* improve SVM performance in a significant way.

The main limitation remains **class imbalance**, especially the tiny Neutral class. Future improvements should focus on:

- Data augmentation
- Reinforcing minority-class representation
- Alternative architectures (e.g., SVM with RBF kernel or transformer-based embeddings)

8: Model Comparison

Model Type		Type	Accuracy	Macro F1	Weighted F1
Logistic Regression	Baseline (Combined Features)		0.8210	0.72	0.83
	GridSearch (Combined Features)		0.8509	0.72	0.85
	GridSearch (TF-IDF Only)		0.8619	0.74	0.86
Multinomial Naive Bayes	Baseline		0.8387	0.40	0.78
	With SMOTE		0.7184	0.47	0.75
	GridSearch-Tuned ($\alpha=0.1$)		0.8463	0.48	0.83

Model Type	Type	Accuracy	Macro F1	Weighted F1
Random Forest	Baseline, class_weight balanced	0.8417	0.41	0.79
	SMOTE + RF	0.8584	0.49	0.83
	RandomizedSearchCV Tuned	0.7732	0.45	0.78
	SMOTE + RF (TF-IDF Best RF)	0.8600	0.48	0.83
XGBoost	Baseline	0.8387	0.42	0.79
	With SMOTE	0.8417	0.46	0.82
Support Vector Machine (SVM)	Baseline (Combined Features)	0.8645	0.52	0.84
	GridSearch (Combined Features)	0.8067	0.50	0.81
	Baseline (TF-IDF Only)	0.8615	0.52	0.84
	GridSearch (TF-IDF Only)	0.8128	0.49	0.81

Key Insights From Our Model Comparison

1 Best Overall Accuracy

- **SVM (Combined Features)** — 86.45%
- **Logistic Regression (TF-IDF, GridSearch)** — 86.19%

2 Best Macro F1 (Fairness Across Classes)

- **SMOTE + Random Forest** — ~0.49
- **Tuned Naive Bayes ($\alpha = 0.1$)** — ~0.48
- **SMOTE + XGBoost** — ~0.46

3 Best Weighted F1 (Majority Performance)

- SVM, Logistic Regression, and Tuned Naive Bayes all score **~0.83–0.86**
→ Strong performance on the dominant **Positive** class.

4 Neutral Class Remains the Weakest

- Recall stays between **0.00–0.10** for all models
- Caused by very few samples (20 total) and overlapping language.

5 SMOTE Trade-Off

- Improves recall for Negative/Neutral
- Lowers overall accuracy by **5–10%**

Final Takeaways

- **Best Overall:** SVM (Combined Features)
- **Best Class-Balanced Model:** SMOTE + Random Forest
- **Best Simple Text Baseline:** Multinomial Naive Bayes ($\alpha = 0.1$)

- **Best Linear Model:** Logistic Regression (TF-IDF Only)

9.0 Business Recommendations

Below are recommendations for our shareholders based on insights from the sentiment analysis we have done.

1. Monitor Product-Specific Issues More Closely

Our analysis shows recurring negative keywords such as **battery**, **need**, **crash**, **store**, **people**, etc.

As shown in the **Negative emotion feature importance table**.

Recommendation

- Set up alerts for sudden spikes in negative keywords.
 - Prioritize fixes for frequently mentioned technical issues (e.g., battery performance) by investing on better way to make the problem go away in future models.
-

2. Speed Up Customer Support for Negative Sentiment

Negative tweets are fewer but highly informative. Delayed responses can damage brand trust.

Recommendation

- Build a dashboard filtering **Negative emotion** tweets in real time.
 - Highlight “high-engagement negative tweets” for quick support replies.
-

3. Leverage Positive Sentiment for Marketing & Branding

The dataset is dominated by positive tweets with strong associations like **ipad**, **link**, **apple**, **google**.

Recommendation

- Amplify themes driving positive engagement.
 - Promote features and events that already generate enthusiasm.
-

4. Improve Detection and Collection of Neutral Tweets

Neutral tweets are extremely underrepresented and are **misclassified by all models**.

Recommendation

- Collect more neutral examples (press releases, news summaries, informational tweets) because this market segment can be potential for increased.

10: Conclusion

This project set out to build an automated sentiment analysis system capable of classifying tweets about Apple and Google products into **positive**, **negative**, and **neutral** categories. Through extensive preprocessing, feature engineering, and experimentation with multiple machine-learning models, the results clearly demonstrate both the potential and limitations of traditional NLP pipelines when applied to highly imbalanced social-media data.

Across all models tested—Logistic Regression, Naive Bayes, Random Forest, XGBoost, and SVM—**the Positive class consistently dominated performance**, reflecting the dataset's heavy skew toward positive sentiment. Models such as **SVM (Combined Features)** and **GridSearch-optimized Logistic Regression** achieved the highest overall accuracy (~86%), showing strong capability in detecting positive sentiment with high precision and recall.

However, the evaluation also highlights a critical challenge: **the Neutral class remains extremely difficult to identify**, with recall often near zero due to its very small sample size and linguistic overlap with other classes. Even advanced imbalance-handling techniques like **SMOTE** only partially improved this issue. While SMOTE-based models (e.g., Random Forest + SMOTE) increased macro-F1 scores—indicating better fairness across classes—they did so at the cost of lower overall accuracy, reflecting the classic trade-off between balanced performance and aggregate correctness.

Despite these limitations, the models demonstrate strong value for real-world stakeholders. The system can reliably detect broad sentiment trends, helping product teams, marketers, and customer-experience analysts monitor brand perception at scale. The insights from feature-importance analysis further reveal the specific vocabulary driving sentiment classifications, providing interpretability that businesses can directly use to understand consumer reactions.

Finally, this project establishes a robust, extensible foundation. The data pipeline, modeling framework, and evaluation strategy can be adapted to more sensitive or socially impactful tasks—such as detecting emotional distress or early signs of depression in social-media text. By mastering the workflow on a commercial sentiment dataset, we prepare the groundwork for developing responsible, human-centered NLP systems in the future.

In summary, the project successfully delivers:

- A working sentiment classification system with strong performance on majority sentiment
- Insights into model behavior under severe class imbalance
- A structured pathway toward improved fairness, interpretability, and future expansion

This aligns closely with industry needs and sets the stage for more advanced NLP applications moving forward.

