Project Title

Measuring Public Perception of Apple and Google on Twitter Using Machine Learning

Phase 1: Business Understanding

Objective

Build a Natural Language Processing (NLP) model to classify tweets mentioning Apple or Google into positive, negative, or neutral sentiments. This will help understand public perception, monitor brand reputation, and inform marketing and product strategies.

Key Questions

- **How do people feel about Apple vs Google products on social media?**
- **Which words or phrases are most influential in expressing sentiment?**
- **Do specific events or product launches trigger changes in sentiment?**

Business Impact

- **Marketing Strategy: Adjust campaigns based on trends in sentiment.**
- **Customer Support: Prioritize responses when negative sentiment spikes.**
- **Product Development: Identify common issues or requests from user feedback.**
- **Crisis Management: Detect negative trends early to mitigate PR risks.**

Phase 2: Data Understanding

Dataset Overview

- **Source: CrowdFlower via data.world**
- **Total tweets: ~9,000 (Apple and Google mentions)**
- **Labels: Positive, Negative, Neutral (human-annotated)**

Key Columns

Column	Description	Example
`tweet_text`	The content of the tweet	"I love my new iPhone!"
`emotion_in_tweet_is_directed_at`	Brand or product mentioned	"iPhone"
`is_there_an_emotion_directed_at_a_brand_or_product`	Sentiment label	Positive, Negative, Neutral

Dataset Suitability

- **Real user opinions, unstructured text**
- **Labeled for sentiment → suitable for supervised ML**
- **Allows tracking trends over time and across products**

Importing Required Libraries

In [28]:

```
# Core & utilities
```

```
# Core & utilities
import time
import re
import joblib
import numpy as np
import pandas as pd

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns
#from wordcloud import WordCloud

# NLP (Natural Language Processing)
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

# Scikit-learn: preprocessing, feature extraction, and model selection
from sklearn.model_selection import train_test_split, GridSearchCV, StratifiedKFold, cros
s_val_score
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer

# Scikit-learn: models
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier

# Scikit-learn: metrics
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    classification_report,
    confusion_matrix
)
from xgboost import XGBClassifier
```

In [29]:

```
# Download NLTK resources (run once)
nltk.download('punkt')
nltk.download('punkt_tab')
nltk.download('stopwords')
nltk.download('wordnet')
```

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

Out[29]:

```
True
```

Loading the data

In [30]:

```
df = pd.read_csv("tweets.csv", encoding="latin-1")
```

Data Quality Assessment

In [31]:

```
#Display the first 5 rows of the dataset
df.head()
```

Out[31]:

	tweet_text	emotion_in_tweet_is_directed_at	is_there_an_emotion_directed_at_a_brand_or_product
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	@sxsw 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 [32]:

```
#Display the names of all available columns in the DataFrame
df.columns
```

Out[32]:

```
Index(['tweet_text', 'emotion_in_tweet_is_directed_at',
       'is_there_an_emotion_directed_at_a_brand_or_product'],
      dtype='object')
```

In [33]:

```
#Display the number of rows and columns in the dataset
df.shape
```

Out[33]:

```
(9093, 3)
```

In [34]:

```
# getting information of the data
df.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
```

In [35]:

```
df['tweet_text'].head()
```

Out[35]:

```
0    .@wesley83 I have a 3G iPhone. After 3 hrs twe...
1    @jessedee Know about @fludapp ? Awesome iPad/i...
2    @swonderlin Can not wait for #iPad 2 also. The...
3    @sxsw I hope this year's festival isn't as cra...
```

```
4    @sxtxstate great stuff on Fri #SXSW: Marissa M...
Name: tweet_text, dtype: object
```

Phase 3: Data Cleaning

In this step, we prepare the dataset by removing unnecessary information, handling missing values, and making the data ready for analysis and modeling. The main steps are:

1 Dropping unwanted columns

Remove irrelevant columns such as emotion_in_tweet_is_directed_at that do not add value to the analysis

Display all column names

This helps as understand:

Which columns contain tweet text

Which column is the sentiment label

Which columns are unnecessary

In [36]:

```
print(df.columns)
```

```
Index(['tweet_text', 'emotion_in_tweet_is_directed_at',
       'is_there_an_emotion_directed_at_a_brand_or_product'],
      dtype='object')
```

In [37]:

```
# DROP UNWANTED COLUMNS
# It does not improve sentiment analysis

df.columns = df.columns.str.strip().str.lower()      # normalize column names

df = df.drop(columns=['emotion_in_tweet_is_directed_at'],
             errors='ignore')
```

2 Handling missing values

Drop rows with missing values in the sentiment column. Fill missing values in other columns with placeholders if necessary.

In [38]:

```
# Checking Missing Values in each column
df.isnull().sum()
```

Out[38]:

```
tweet_text                                            1
is_there_an_emotion_directed_at_a_brand_or_product    0
dtype: int64
```

In [39]:

```
# HANDLE MISSING VALUES
# To avoid Breaking the model training
# Create wrong predictions

# Drop missing sentiment rows
```

```
df = df.dropna(subset=['is_there_an_emotion_directed_at_a_brand_or_product'])


# Fill other missing values
# We replace them with "Unknown" to avoid errors:
# Tf-IDF Does not allow NaN
df.fillna("Unknown", inplace=True)
```

3 Renaming columns and sentiment categories

Rename long column names to shorter, more intuitive ones (e.g., is_there_an_emotion_directed_at_a_brand_or_product → sentiment). Standardize sentiment labels (No emotion toward brand or product → neutral, etc.).

The sentiment column name is too long and hard to type

Code becomes messy

Models prefer simple, readable column names

In [40]:

```
# RENAME COLUMNS + STANDARDIZE SENTIMENT LABELS

df.rename(columns={'is_there_an_emotion_directed_at_a_brand_or_product': 'sentiment'},
          inplace=True)

# Inspect Sentiment Values
print(df['sentiment'].unique())
```

```
['Negative emotion' 'Positive emotion'
 'No emotion toward brand or product' "I can't tell"]
```

We notice

Some labels are UPPERCASE, some lowercase

Some have spaces like "negative Emotion "

“I can’t tell” is NOT useful for model training

“No emotion…” is too long

In [41]:

```
#Why standardize sentiment
# For Consistency,Easier coding,Better accuracy
# First convert to lowercase
df['sentiment'] = df['sentiment'].astype(str).str.lower().str.strip()
# Replace long labels with cleaner ones:
df['sentiment'] = df['sentiment'].replace({
    'no emotion toward brand or product': 'neutral',
    'positive emotion': 'positive',
    'negative emotion': 'negative',
    "i can't tell": 'unknown'
})
```

4 Removing unwanted categories

Drop rows where the sentiment is I can't tell since it does not provide useful information for the model.

In [42]:

```
# Chech which cell is empty
# df.isnull().sum()
```

```
# If a cell is empty, it marks it as True(missing)
# If it is not empty, it marks it as False(not missing)

# STEP 4: REMOVE UNWANTED CATEGORIES
df = df[df['sentiment'] != 'unknown']
```

5 Cleaning text data

Convert text to lowercase. Remove URLs, numbers, punctuation, and extra spaces. Ensure the text is consistent and ready for vectorization.

In [43]:

```
#Raw text from tweets is messy
#URLs like http://...
#Mentions like @username
#Hashtags like #happy
#Numbers, punctuation, extra spaces
print(df['tweet_text'].head())
```

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

In [44]:

```
# CLEAN TEXT DATA
# Define a cleaning function

def clean_text(text):
    text = str(text).lower()

    text = re.sub(r'http\S+|www\S+', '', text)      # remove URLs
    text = re.sub(r'@[A-Za-z0-9_]+', '', text)       # remove mentions
    text = re.sub(r'#[A-Za-z0-9_]+', '', text)       # remove hashtags
    text = re.sub(r'[^a-z\s]', '', text)             # remove punctuation + numbers
    text = re.sub(r'\s+', ' ', text).strip()         # remove extra spaces

    return text
#Apply cleaning function
df['text_clean'] = df['tweet_text'].apply(clean_text)

# Remove stopwords
#Stopwords like "the", "is", "at" don't help the model.
stop_words = set(stopwords.words('english'))
df['text_clean'] = df['text_clean'].apply(
    lambda x: ' '.join([w for w in x.split() if w not in stop_words])
)

# Tokenize
# Split each tweet into words (tokens) for further processing:
df['tokens'] = df['text_clean'].apply(word_tokenize)

# Lemmatize
#Convert words to their root form, e.g., "running" → "run"
lemmatizer = WordNetLemmatizer()
df['tokens_lemmatized'] = df['tokens'].apply(
    lambda t: [lemmatizer.lemmatize(w) for w in t]
)

# Rebuild final text
# After lemmatization, join tokens back into a clean string for vectorization:
df['text_final'] = df['tokens_lemmatized'].apply(lambda t: " ".join(t))
```

6 Text vectorization

Transform the cleaned text into numerical format using techniques like: Bag of Words (CountVectorizer) Counts how many times each word appears in a document

TF-IDF (TfidfVectorizer) Counts words but reduces weight of common words across all documents

Because Machine learning models cannot work directly with text.

Vectorization converts cleaned text into numbers that a model can understand.

In [45]:

```
# TEXT VECTORIZATION

# Bag of Words
#Creates a vocabulary of all words in the dataset
#Counts how many times each word appears in each tweet
#Returns a sparse matrix (numerical format)
bow = CountVectorizer()
X_bow = bow.fit_transform(df['text_final'])

# TF-IDF (recommended)
#Words that appear in many tweets (like "love", "product") get lower weight
#Words that are rare but important get higher weight
#Makes features more meaningful for classification
tfidf = TfidfVectorizer()
X_tfidf = tfidf.fit_transform(df['text_final'])


# Features and labels
# X → what the model uses to learn patterns
# y → what the model tries to predict (positive/neutral/negative)

X = X_tfidf # Features (numerical representation of text)
y = df['sentiment']#Target labels
```

In [46]:

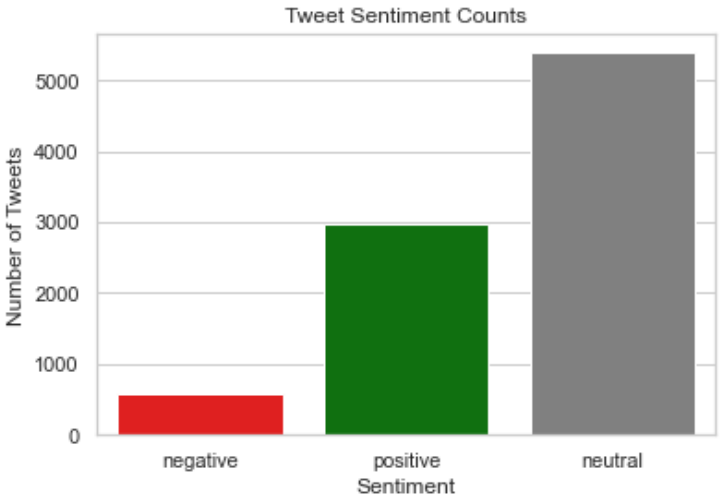
```
# Chicking the final output
print(df[['tweet_text', 'sentiment', 'text_final']].head())
print("\nSentiment distribution:\n", y.value_counts())
```

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

                                          text_final
0  g iphone hr tweeting dead need upgrade plugin ...
1  know awesome ipadiphone app youll likely appre...
2                                  wait also sale
3     hope year festival isnt crashy year iphone app
4  great stuff fri marissa mayer google tim oreil...

Sentiment distribution:
 neutral     5389
positive    2978
negative     570
Name: sentiment, dtype: int64
```

In [47]:

```
import seaborn as sns

plt.figure(figsize=(6,4))
sns.countplot(x='sentiment', data=df, palette={'neutral':'gray', 'positive':'green', 'ne
gative':'red'})
plt.title('Tweet Sentiment Counts')
plt.xlabel('Sentiment')
```

```
plt.ylabel('Number of Tweets')
plt.show()
```



Interpretation of Tweet Sentiment Counts (Bar Chart)

- **Neutral (5,500 tweets):**
  The majority of tweets are neutral, indicating that most users are either sharing information or not expressing strong opinions about Apple or Google products.
- **Positive (3,000 tweets):**
  A significant portion of tweets show favorable sentiment, reflecting user satisfaction or approval.
- **Negative (600 tweets):**
  Only a small fraction of tweets are negative, suggesting limited dissatisfaction expressed on Twitter.

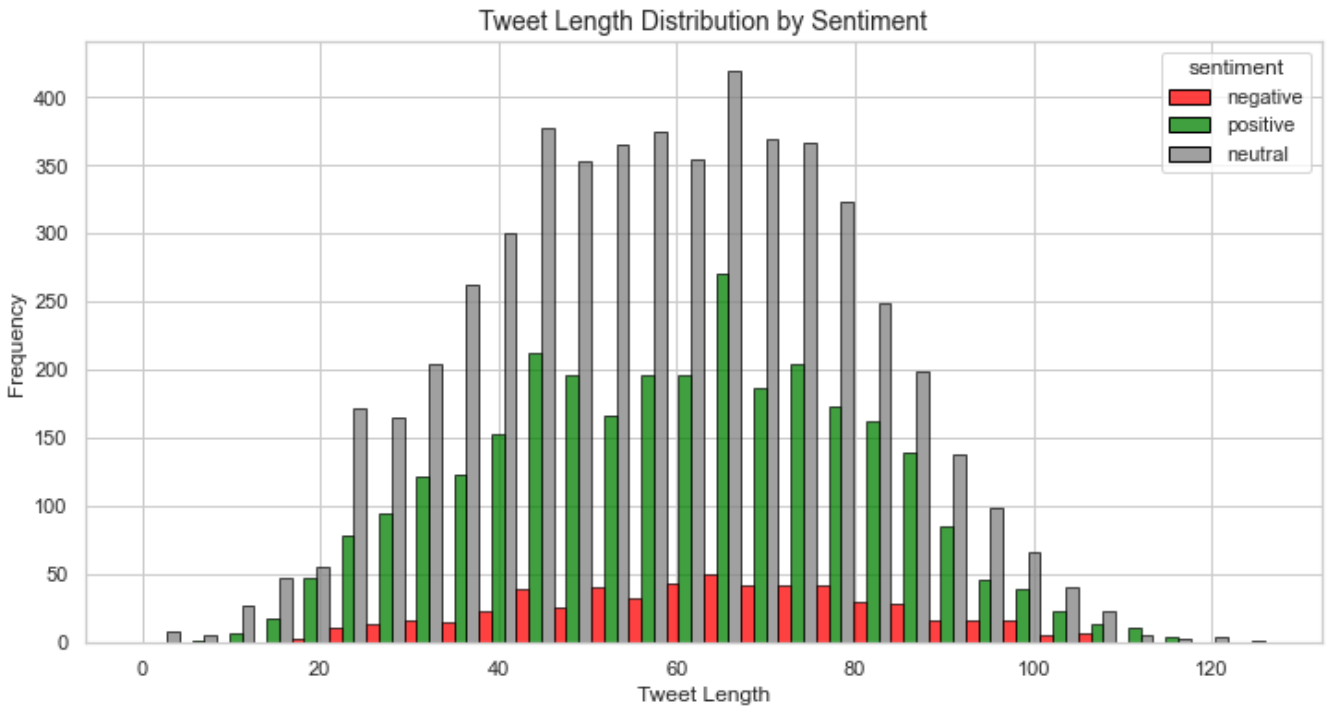
Indeed The bar chart confirms that the majority of users are neutral or positive, with negative reactions being minimal. This insight can guide marketing and customer engagement strategies to maintain positive sentiment while monitoring negative feedback for potential improvements.

Analyzing Tweet Lengths by Sentiment

In [48]:

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

# Ensure tweet text is string and drop missing values
df['text_final'] = df['text_final'].dropna().astype(str)

# Create a new column with tweet lengths
df['tweet_length'] = df['text_final'].str.len()

# Set plot style
sns.set(style="whitegrid")

# Define custom colors for each sentiment
custom_palette = {
    'positive': 'green',
    'negative': 'red',
    'neutral': 'gray'
}

# Plot histogram of tweet lengths by sentiment
plt.figure(figsize=(12, 6))
sns.histplot(
    data=df,
    x='tweet_length',
    hue='sentiment',         # Different colors for each sentiment
    bins=30,
    palette=custom_palette,    # Apply custom colors
```

```
    multiple='dodge',        # Show side-by-side bars for comparison
    edgecolor='black'
)

# Add labels and title
plt.xlabel('Tweet Length', fontsize=12)
plt.ylabel('Frequency', fontsize=12)
plt.title('Tweet Length Distribution by Sentiment', fontsize=14)
plt.show()
```



Interpretation of Tweet Length Distribution by Sentiment

The histogram above shows how tweet lengths vary across different sentiment categories: positive, negative, and neutral.

Positive Tweets:
These tweets tend to have moderate to slightly longer lengths, mostly between 50 and 100 characters.
This suggests that users expressing positive emotions often write a bit more, possibly sharing detailed feedback, praise, or experiences.

Negative Tweets:
Negative tweets are generally shorter, clustering around 20 to 80 characters.
This indicates that users expressing dissatisfaction often post brief, direct messages, likely reflecting frustration or immediate reactions.

Neutral Tweets:
Neutral tweets show a broader range of lengths, including some of the longest tweets in the dataset.
These are likely factual, informational, or descriptive messages that require more words to convey context without conveying emotion.

Overall Patterns:

- **Positive and neutral tweets are generally longer than negative ones.**
- **Negative tweets are concise, often signaling quick reactions or complaints.**

This insight can be useful for feature engineering, as tweet length might help predict sentiment when combined with other text-based features.

Phase 4 .Modeling(Multiclass Sentiment Classification)

Model 1 Logistic Regression (Baseline) Text +Numerical Features

Why Logistic Regression?
Handles Multiclass Problems (negative, neutral, positive)

Trains and predicts quickly

Works great with text data(TF-IDF)("great", "amazing" → positive weight)("terrible", "worst" → negative weight)

Has built-in regularization to prevent overfitting

Interpretable results

Handles Imbalanced Data

Probability outputs

1. **STEP 1: Create Numeric Features**

Negative tweets might be shorter and angrier: "This sucks!" Positive tweets might be longer with more enthusiasm: "I absolutely love this amazing product!!!"

In [49]:

```
#Counting how many characters are in each tweet
#Counting how many words are in each tweet

df['tweet_length'] = df['text_final'].str.len()
df['num_words'] = df['text_final'].str.split().map(len)

numeric_features = ['tweet_length', 'num_words']
```

2 Prepare the Data

X = "What the model sees"

y = "What the model should predict"

In [50]:

```
# X = our input features (text + numbers)
# y = what we want to predict (sentiment labels)
X = df[['text_final'] + numeric_features]
y = df['sentiment']  # Keep as text labels (negative/neutral/positive)
```

3 Split Data into Training & Testing

You can't test a student using the same questions they studied from! We need "unseen" data to see if the model truly understands

In [51]:

```
# We trained on 80% of data, tested on 20% to see if model works on NEW data
# stratify=y ensures we keep the same proportion of each sentiment in both sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

1. **Build Preprocessing Pipeline Why Pipeline ?**

Converts text to numbers (models only understand numbers) Scales features equally (prevents bias toward large numbers) Automates everything (one command does all preprocessing) Prevents mistakes (no data leakage, consistent processing) Makes code clean (easy to read, maintain, deploy) Works with GridSearchCV (proper cross-validation)

In [52]:

```
#Text and numbers need different processing:
```

```
#Text: Convert to TF-IDF vectors (finds important words)
#Numbers: Scale to similar ranges (StandardScaler)
preprocessor = ColumnTransformer([
    ('tfidf', TfidfVectorizer(max_features=5000, ngram_range=(1,2), stop_words='english'
),
     'text_final'),  # Process text column
    ('num', StandardScaler(), numeric_features)  # Process numeric columns
])
```

5: Create the Full Pipeline

Combines everything into ONE object:

First: Preprocess the data (TF-IDF + scaling) Then: Feed it to Logistic Regression

In [53]:

```
#class_weight='balanced' → Handles imbalanced data (if you have 100 positive but only 10
negative tweets, this makes sure the model doesn't ignore negatives)
#solver='saga' → The algorithm used to find the best model
#max_iter=2000 → Try up to 2000 times to find the best solution
pipe = Pipeline([
    ('preprocess', preprocessor),
    ('clf', LogisticRegression(solver='saga',
                               max_iter=2000,
                               class_weight='balanced',  # Handles imbalanced data
                               random_state=42))
])
```

1. **Hyperparameter Tuning**

We don't know the BEST settings, so we try different combinations! What's C?

Regularization strength (prevents overfitting) Small C (0.01) → Simple model, might underfit Large C (5) → Complex model, might overfit We're testing: 0.01, 0.1, 1, and 5

In [54]:

```
#l1 → Makes some features exactly zero (feature selection)
#l2 → Shrinks all features a little bit
#We're testing both
#Total combinations: 4 C values × 2 penalties = 8 combinations

param_grid = {
    'clf__C': [0.01, 0.1, 1, 5],
    'clf__penalty': ['l1', 'l2']
}

# 5-fold cross-validation: splits training data into 5 parts,
# trains on 4, validates on 1, repeats 5 times
#More reliable than a single train/test split Ensures the model works
#well on different subsets of data
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# GridSearchCV tries all combinations and picks the best
grid = GridSearchCV(
    pipe,
    param_grid,
    scoring='f1_macro',  # Metric that treats all classes equally
    cv=cv,
    n_jobs=-1,  # Use all CPU cores
    verbose=2
)
```

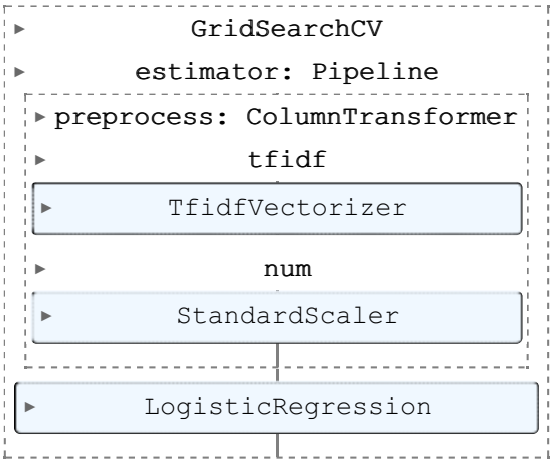
1. **Train The Model**

In [55]:

```
grid.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 8 candidates, totalling 40 fits
```

Out[55]:



1. **Get best model and predict**

   Retrieves the winning model (C=5, penalty=l2) Uses it to predict sentiments on the TEST set (the 20% we held back)

In [56]:

```
best_model = grid.best_estimator_
y_pred = best_model.predict(X_test)
```

9.Evaluate The Performance

In [57]:

```
print("=" * 60)
print("BEST MODEL RESULTS")
print("=" * 60)
print(f"Best Parameters: {grid.best_params_}")
print(f"Best CV F1 Score: {grid.best_score_:.4f}")
print(f"\nTest Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print(f"Test Macro F1 Score: {f1_score(y_test, y_pred, average='macro'):.4f}")
print("\n" + "=" * 60)
print("CLASSIFICATION REPORT")
print("=" * 60)
print(classification_report(y_test, y_pred))
```

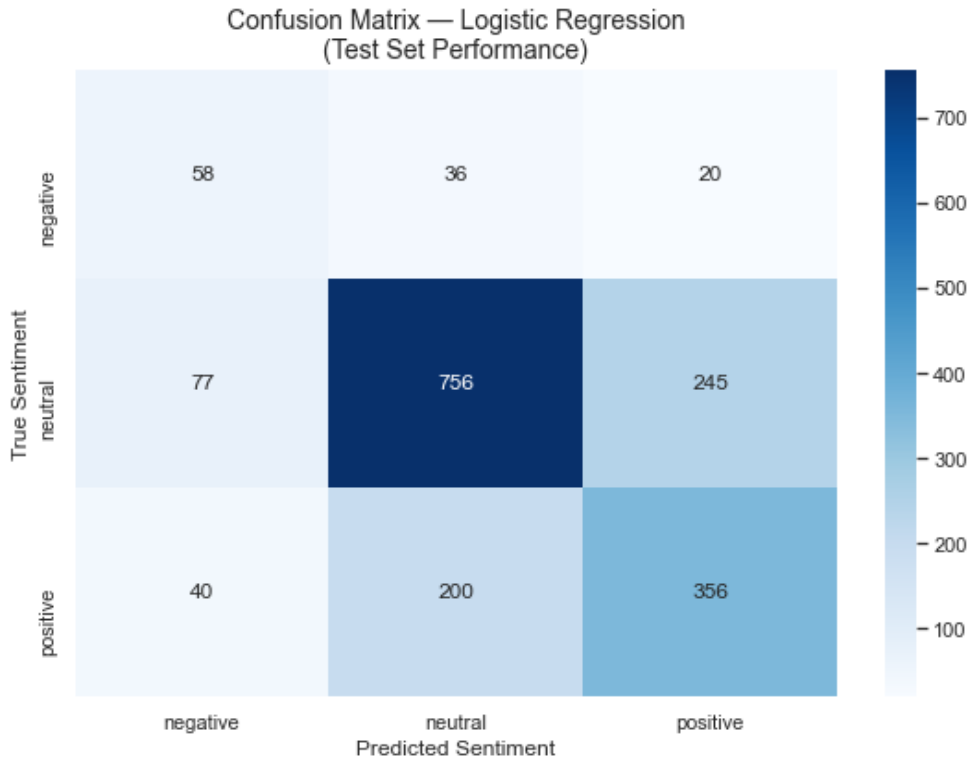
```
============================================================
BEST MODEL RESULTS
============================================================
Best Parameters: {'clf__C': 5, 'clf__penalty': 'l2'}
Best CV F1 Score: 0.5495

Test Accuracy: 0.6544
Test Macro F1 Score: 0.5723

============================================================
CLASSIFICATION REPORT
============================================================
              precision    recall  f1-score   support

    negative       0.33      0.51      0.40       114
     neutral       0.76      0.70      0.73      1078
    positive       0.57      0.60      0.59       596

    accuracy                           0.65      1788
   macro avg       0.56      0.60      0.57      1788
weighted avg       0.67      0.65      0.66      1788
```

10.Visualize Confusion Matrix

Shows where the model makes mistakes Diagonal = correct predictions Off-diagonal = mistakes (e.g., predicted positive but was negative)

In [58]:

```
# Get sentiment labels in sorted order for consistent display
sentiment_labels = sorted(y.unique())

# Create confusion matrix with explicit label ordering
cm = confusion_matrix(y_test, y_pred, labels=sentiment_labels)

# Plot
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=sentiment_labels,  # FIXED: Show actual sentiment names
            yticklabels=sentiment_labels)  # FIXED: Show actual sentiment names
plt.xlabel("Predicted Sentiment", fontsize=12)
plt.ylabel("True Sentiment", fontsize=12)
plt.title("Confusion Matrix — Logistic Regression\n(Test Set Performance)", fontsize=14)
plt.tight_layout()
plt.show()


#  ANALYZE RESULTS
print("\n" + "=" * 60)
print("INTERPRETATION")
print("=" * 60)
print(f"Total test samples: {len(y_test)}")
print(f"Correct predictions: {(y_test == y_pred).sum()}")
print(f"Incorrect predictions: {(y_test != y_pred).sum()}")
print("\nPer-class breakdown:")
for label in sentiment_labels:
    count = (y_test == label).sum()
    correct = ((y_test == label) & (y_pred == label)).sum()
    print(f"  {label}: {correct}/{count} correct ({correct/count*100:.1f}%)")
```




```
============================================================
INTERPRETATION
============================================================
Total test samples: 1788
Correct predictions: 1170
Incorrect predictions: 618
```

```
Per-class breakdown:
  negative: 58/114 correct (50.9%)
  neutral: 756/1078 correct (70.1%)
  positive: 356/596 correct (59.7%)
```

MODEL 2 Random Forest Classifier (TF-IDF + Numeric Features)

Why Random Forest?

Handles nonlinear patterns

Works well with mixed features (TF-IDF + numeric)

More robust to noise

Naturally reduces overfitting via bagging

Step 1: Encode Sentiment Labels

Random Forest cannot work with text labels ("positive", "neutral", "negative"). We convert them into numbers using LabelEncoder():

negative → 0

neutral → 1

positive → 2

Because Random Forest only understands numbers, not text.

In [59]:

```
le = LabelEncoder()

y_train_enc = le.fit_transform(y_train)    # Convert training labels
y_test_enc = le.transform(y_test)          # Convert test labels
```

2: Apply the SAME Preprocessor (TF-IDF + Scaling)

We must use the same text+numeric preprocessing that Logistic Regression used, to make results comparable.

Converts text into TF-IDF vectors

Standardizes numeric features (tweet_length, num_words)

Ensures RF model sees the same transformed features

In [60]:

```
# Transform text + numeric features
preprocessor.fit(X_train)                 # Learn vocabulary + scalers

X_train_tfidf = preprocessor.transform(X_train)    # Apply transformations
X_test_tfidf = preprocessor.transform(X_test)
```

3: Initialize Random Forest Model

We create a Random Forest with:

200 trees (n_estimators=200)

class_weight='balanced' (fixes class imbalance)

random_state=42 (reproducibility)

n_jobs=-1 (use all CPU cores)

It builds many decision trees and averages them → more stable predictions.

In [61]:

```
# Initialize Random Forest model

rf_classifier = RandomForestClassifier(
    n_estimators=200,        # Number of trees
    random_state=42,
    class_weight='balanced', # Helps handle imbalanced classes
    n_jobs=-1                # Use all CPU cores
)
```

4: Train the Model

We train Random Forest on the transformed training data.

Each tree learns patterns in TF-IDF + numeric features

Model learns what words/sentences correspond to each sentiment

In [62]:

```
# Train the model
rf_classifier.fit(X_train_tfidf, y_train_enc)
```

Out[62]:

```
RandomForestClassifier(class_weight='balanced', n_estimators=200, n_jobs=-1,
                       random_state=42)
```

5: Make Predictions

We ask the model to predict sentiments for the test set.

To check how well it performs on data it has never seen before.

In [63]:

```
# Predictions on test data
y_pred_rf = rf_classifier.predict(X_test_tfidf)
```

6: Evaluate the Model

We calculate:

Accuracy

Macro F1 score

Detailed classification report (precision, recall, f1 for each class)

Macro F1 = best metric for imbalanced sentiment data

In [64]:

```
#  Evaluation


acc_rf = accuracy_score(y_test_enc, y_pred_rf)
f1_rf = f1_score(y_test_enc, y_pred_rf, average='macro')

print(f"\nRandom Forest Test Accuracy: {acc_rf:.4f}")
print(f"Random Forest Macro F1 Score: {f1_rf:.4f}")
```

```
print("\nClassification Report — Random Forest:\n")
print(classification_report(y_test_enc, y_pred_rf, target_names=le.classes_))
```

```
Random Forest Test Accuracy: 0.6734
Random Forest Macro F1 Score: 0.5204

Classification Report — Random Forest:

              precision    recall  f1-score   support

    negative       0.58      0.19      0.29       114
     neutral       0.69      0.86      0.77      1078
    positive       0.62      0.42      0.50       596

    accuracy                           0.67      1788
   macro avg       0.63      0.49      0.52      1788
weighted avg       0.66      0.67      0.65      1788
```

7: Confusion Matrix Visualization

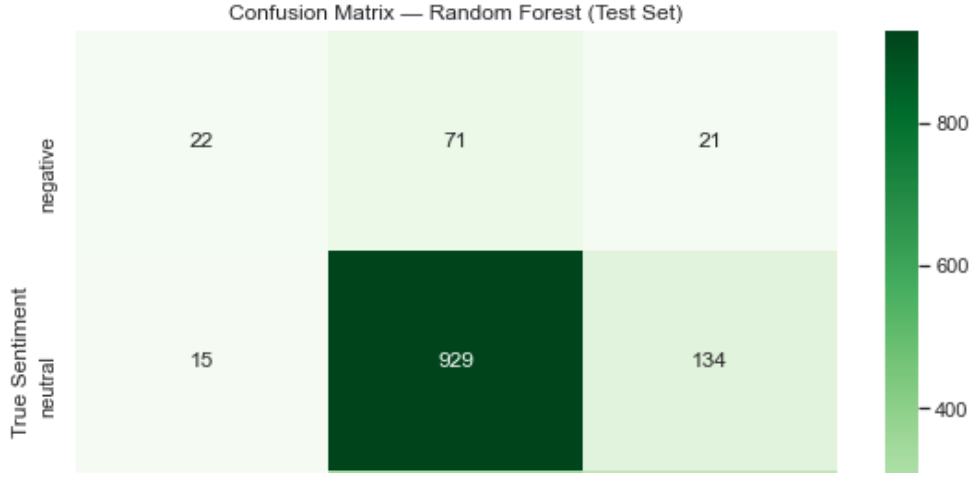
Helps you see:

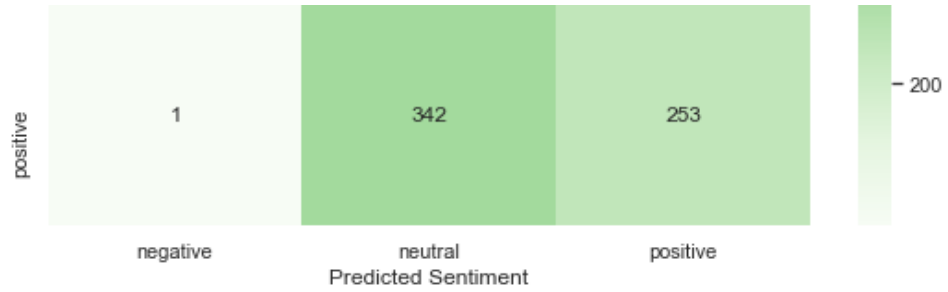
How many correct predictions (diagonal)

Where the model confuses classes (off-diagonal)

In [65]:

```
cm_rf = confusion_matrix(y_test_enc, y_pred_rf)

plt.figure(figsize=(8, 6))
sns.heatmap(cm_rf, annot=True, fmt='d', cmap='Greens',
            xticklabels=le.classes_, yticklabels=le.classes_)

plt.xlabel("Predicted Sentiment")
plt.ylabel("True Sentiment")
plt.title("Confusion Matrix — Random Forest (Test Set)")
plt.tight_layout()
plt.show()

#  ANALYZE RESULTS
print("\n" + "="*60)
print("INTERPRETATION — Random Forest")
print("="*60)

print(f"Total test samples: {len(y_test_enc)}")
print(f"Correct predictions: {(y_test_enc == y_pred_rf).sum()}")
print(f"Incorrect predictions: {(y_test_enc != y_pred_rf).sum()}")

print("\nPer-class breakdown:")
for idx, label in enumerate(le.classes_):
    total = (y_test_enc == idx).sum()
    correct = ((y_test_enc == idx) & (y_pred_rf == idx)).sum()
    print(f"  {label}: {correct}/{total} correct ({correct/total*100:.2f}%)")
```






```
============================================================
INTERPRETATION — Random Forest
============================================================
Total test samples: 1788
Correct predictions: 1204
Incorrect predictions: 584

Per-class breakdown:
  negative: 22/114 correct (19.30%)
  neutral: 929/1078 correct (86.18%)
  positive: 253/596 correct (42.45%)
```

MODEL 3 SVM Classifier (TF-IDF + Numeric Features)

SVM

Works well for high-dimensional data like text (TF-IDF vectors).

Linear kernel is a good baseline for text classification.

Handles imbalanced classes with class_weight='balanced'.

Can produce clear margins between sentiment classes.

1: Encode Sentiment Labels

SVM cannot work with text labels. We convert them to numbers:

In [66]:

```
#negative → 0, neutral → 1, positive → 2
le = LabelEncoder()

y_train_enc = le.fit_transform(y_train)    # Convert training labels
y_test_enc = le.transform(y_test)          # Convert test labels
```

2: Apply the SAME Preprocessor (TF-IDF + Scaling)

We ensure SVM sees the same transformed features as Random Forest:

In [67]:

```
# Transform text + numeric features
preprocessor.fit(X_train)                 # Learn vocabulary + scalers
X_train_tfidf = preprocessor.transform(X_train)    # Apply transformations
X_test_tfidf = preprocessor.transform(X_test)
```

3: Initialize SVM Model

Linear kernel is usually effective for text classification.

class_weight='balanced' helps when some classes are smaller.

In [68]:

```
svm_classifier = SVC(
```

```
    kernel='linear',          # Linear kernel for text
    class_weight='balanced', # Handle class imbalance
    random_state=42
)
```

4: Train the Model

In [69]:

```
#Model learns patterns in TF-IDF + numeric features.

#Finds optimal hyperplane to separate sentiment classes.
svm_classifier.fit(X_train_tfidf, y_train_enc)
```

Out[69]:

SVC

```
SVC(class_weight='balanced', kernel='linear', random_state=42)
```

5: Make Predictions

Predicts sentiment for the test set.

In [70]:

```
y_pred_svm = svm_classifier.predict(X_test_tfidf)
```

6: Evaluate the Model

Macro F1 is key for imbalanced sentiment datasets.

Classification report gives precision, recall, f1 per class.

In [71]:

```
acc_svm = accuracy_score(y_test_enc, y_pred_svm)
f1_svm = f1_score(y_test_enc, y_pred_svm, average='macro')

print(f"\nSVM Test Accuracy: {acc_svm:.4f}")
print(f"SVM Macro F1 Score: {f1_svm:.4f}")

print("\nClassification Report — SVM:\n")
print(classification_report(y_test_enc, y_pred_svm, target_names=le.classes_))
```

```
SVM Test Accuracy: 0.6437
SVM Macro F1 Score: 0.5663

Classification Report — SVM:

              precision    recall  f1-score   support

    negative       0.31      0.54      0.40       114
     neutral       0.76      0.68      0.72      1078
    positive       0.57      0.60      0.59       596

    accuracy                           0.64      1788
   macro avg       0.55      0.61      0.57      1788
weighted avg       0.67      0.64      0.65      1788
```

In [72]:

```
#Diagonal: correct predictions.

#Off-diagonal: class confusion.
#Confusion Matrix Visualization
cm_svm = confusion_matrix(y_test_enc, y_pred_svm)
```

```
plt.figure(figsize=(8, 6))
sns.heatmap(cm_svm, annot=True, fmt='d', cmap='Oranges',
            xticklabels=le.classes_, yticklabels=le.classes_)
plt.xlabel("Predicted Sentiment")
plt.ylabel("True Sentiment")
plt.title("Confusion Matrix — SVM (Test Set)")
plt.tight_layout()
plt.show()


#Analyze Results
print("\n" + "="*60)
print("INTERPRETATION — SVM")
print("="*60)

print(f"Total test samples: {len(y_test_enc)}")
print(f"Correct predictions: {(y_test_enc == y_pred_svm).sum()}")
print(f"Incorrect predictions: {(y_test_enc != y_pred_svm).sum()}")

print("\nPer-class breakdown:")
for idx, label in enumerate(le.classes_):
    total = (y_test_enc == idx).sum()
    correct = ((y_test_enc == idx) & (y_pred_svm == idx)).sum()
    print(f"  {label}: {correct}/{total} correct ({correct/total*100:.2f}%)")
```


```
============================================================
INTERPRETATION — SVM
============================================================
Total test samples: 1788
Correct predictions: 1151
Incorrect predictions: 637

Per-class breakdown:
  negative: 62/114 correct (54.39%)
  neutral: 729/1078 correct (67.63%)
  positive: 360/596 correct (60.40%)
```

MODEL 4 XGBoost Classifier (TF-IDF + Numeric Features)

Why XGBoost?

Gradient boosting algorithm → handles nonlinear relationships.

Works well with mixed features (TF-IDF + numeric features).

Handles class imbalance via hyperparameters or sample weighting.

Often achieves higher accuracy than simpler models like Naïve Bayes or linear SVM.

1: Encode Sentiment Labels Mapping example: negative → 0, neutral → 1, positive → 2

In [73]:

```
le = LabelEncoder()
y_train_enc = le.fit_transform(y_train)    # Convert training labels
y_test_enc = le.transform(y_test)          # Convert test labels
```

2: Apply the SAME Preprocessor (TF-IDF + Numeric Features)

Even though NB doesn't require scaling, you still need TF-IDF + numeric features for consistency with other models:

In [74]:

```
# Transform text + numeric features
preprocessor.fit(X_train)
X_train_tfidf = preprocessor.transform(X_train)
X_test_tfidf = preprocessor.transform(X_test)
```

3: Initialize XGBoost Model

n_estimators → number of trees

max_depth → controls complexity of each tree

eval_metric='mlogloss' → appropriate for multiclass classification

In [75]:

```
xgb_classifier = XGBClassifier(
    n_estimators=200,       # Number of boosting rounds
    learning_rate=0.1,      # Step size shrinkage
    max_depth=6,            # Maximum depth of a tree
    random_state=42,        # For reproducibility
    eval_metric='mlogloss' # Multiclass log loss
)
```

4: Train the Model

Model learns patterns from TF-IDF + numeric features.

In [76]:

```
xgb_classifier.fit(X_train_tfidf, y_train_enc)
```

Out[76]:

```
▾                                XGBClassifier
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, eval_metric='mlogloss',
              gamma=0, gpu_id=-1, importance_type='gain',
              interaction_constraints='', learning_rate=0.1, max_delta_step=0
,
              max_depth=6, min_child_weight=1, missing=nan,
              monotone_constraints='()', n_estimators=200, n_jobs=0,
              num_parallel_tree=1, objective='multi:softprob', random_state=4
2,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=None, subsample=1,
```

5: Make Predictions Predicts sentiment for unseen test data

5. Make Predictions Predicts sentiment for unseen test data.

In [77]:

```
y_pred_xgb = xgb_classifier.predict(X_test_tfidf)
```

6: Evaluate the Model

In [78]:

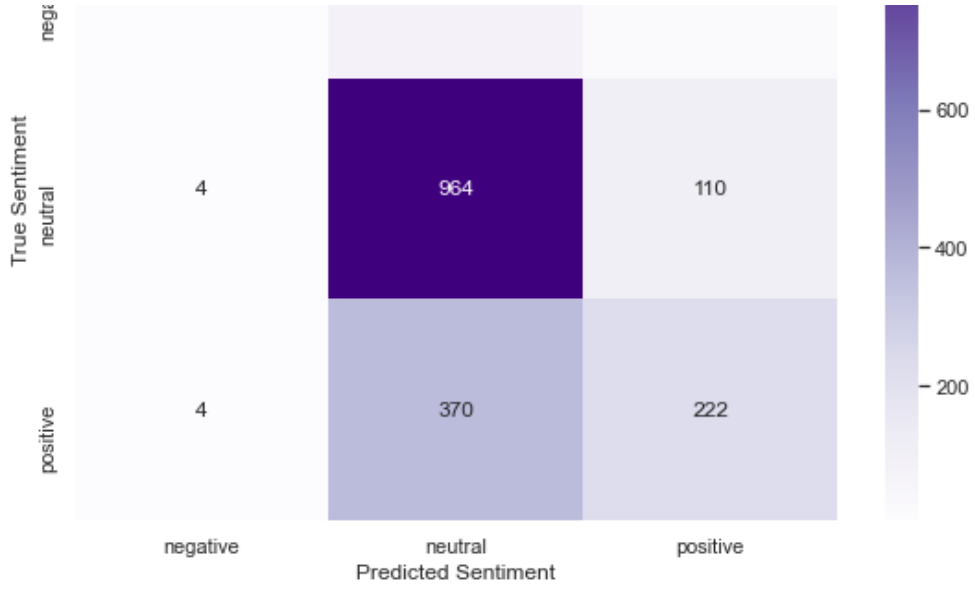
```
acc_xgb = accuracy_score(y_test_enc, y_pred_xgb)
f1_xgb = f1_score(y_test_enc, y_pred_xgb, average='macro')

print(f"\nXGBoost Test Accuracy: {acc_xgb:.4f}")
print(f"XGBoost Macro F1 Score: {f1_xgb:.4f}")

print("\nClassification Report — XGBoost:\n")
print(classification_report(y_test_enc, y_pred_xgb, target_names=le.classes_))
```

```
XGBoost Test Accuracy: 0.6695
XGBoost Macro F1 Score: 0.4687

Classification Report — XGBoost:

              precision    recall  f1-score   support

    negative       0.58      0.10      0.17       114
     neutral       0.68      0.89      0.77      1078
    positive       0.63      0.37      0.47       596

    accuracy                           0.67      1788
   macro avg       0.63      0.45      0.47      1788
weighted avg       0.66      0.67      0.63      1788
```

Confusion Matrix Visualization

In [79]:

```
cm_xgb = confusion_matrix(y_test_enc, y_pred_xgb)

plt.figure(figsize=(8,6))
sns.heatmap(cm_xgb, annot=True, fmt='d', cmap='Purples',
            xticklabels=le.classes_, yticklabels=le.classes_)
plt.xlabel("Predicted Sentiment")
plt.ylabel("True Sentiment")
plt.title("Confusion Matrix — XGBoost (Test Set)")
plt.tight_layout()
plt.savefig("CMxgb_multclass.png", dpi=300, bbox_inches='tight')
plt.show()
# Analyze the results
print("\n" + "="*60)
print("INTERPRETATION — XGBoost")
print("="*60)

print(f"Total test samples: {len(y_test_enc)}")
print(f"Correct predictions: {(y_test_enc == y_pred_xgb).sum()}")
print(f"Incorrect predictions: {(y_test_enc != y_pred_xgb).sum()}")

print("\nPer-class breakdown:")
for idx, label in enumerate(le.classes_):
    total = (y_test_enc == idx).sum()
    correct = ((y_test_enc == idx) & (y_pred_xgb == idx)).sum()
    print(f"  {label}: {correct}/{total} correct ({correct/total*100:.2f}%)")
```




```
============================================================
INTERPRETATION — XGBoost
============================================================
Total test samples: 1788
Correct predictions: 1197
Incorrect predictions: 591

Per-class breakdown:
  negative: 11/114 correct (9.65%)
  neutral: 964/1078 correct (89.42%)
  positive: 222/596 correct (37.25%)
```

Phase 5: Modeling Interpretation (Multiclass Sentiment Classification)

1 Overall Comparison of Models

Model	Test Accuracy	Macro F1	Observation
Logistic Regression	0.6544	0.5723	Baseline model. Performs fairly balanced across classes.
Random Forest	0.6734	0.5204	Accuracy slightly higher, but macro F1 lower → struggles on minority classes (Negative).
SVM	0.6437	0.5663	Similar to Logistic Regression; better recall on Negative.
XGBoost	0.6695	0.4687	Accuracy okay, but macro F1 low → very poor on Negative and Positive classes.

Observation:

- **Accuracy can be misleading due to class imbalance (Neutral dominates).**
- **Macro F1 is a better metric as it gives equal weight to all classes.**

---

2 Per-Class Performance

Negative Tweets (114 samples)

Model	Precision	Recall	F1	Correct Predictions
Logistic Regression	0.33	0.51	0.40	58
Random Forest	0.58	0.19	0.29	22
SVM	0.31	0.54	0.40	62
XGBoost	0.58	0.10	0.17	11

Interpretation:

- Negative tweets are hard to predict due to minority class.
- Logistic Regression and SVM provide better **recall** → fewer missed negatives.
- Random Forest and XGBoost show high precision but very low recall → miss most negative tweets.

Neutral Tweets (1078 samples)

Model	Precision	Recall	F1	Correct Predictions
Logistic Regression	0.76	0.70	0.73	756
Random Forest	0.69	0.86	0.77	929
SVM	0.76	0.68	0.72	729
XGBoost	0.68	0.89	0.77	964

Interpretation:

- All models perform well on Neutral due to majority class.
- Random Forest & XGBoost maximize Neutral recall but this comes at the expense of minority classes.

Positive Tweets (596 samples)

Model	Precision	Recall	F1	Correct Predictions
Logistic Regression	0.57	0.60	0.59	356
Random Forest	0.62	0.42	0.50	253
SVM	0.57	0.60	0.59	360
XGBoost	0.63	0.37	0.47	222

Interpretation:

- Positive tweets are also challenging.
- Logistic Regression and SVM have the **highest F1**.
- Random Forest & XGBoost favor Neutral → lower recall on Positive.

---

3 Confusion Matrix Insights

- **Diagonal values** = correctly predicted samples.
- **Off-diagonal values** → common confusions:
  - Negative → often predicted as Neutral
  - Positive → sometimes predicted as Neutral
  - Neutral → usually predicted correctly (large diagonal)

Takeaway:

- Models tend to **overpredict the majority class (Neutral)**.
- Logistic Regression and SVM are more **balanced across classes**.
- XGBoost & Random Forest show **bias toward the majority class**.

---

4 Overall Interpretation

- **Best models for minority classes (Negative, Positive):** Logistic Regression & SVM
- **Best for majority class (Neutral):** XGBoost & Random Forest
- **Best overall balance (Macro F1):** Logistic Regression (0.5723)
- **Key challenge:** Class imbalance causes models to favor Neutral, hurting minority classes.

---

5 Recommendations Based on Models

1. Use **Logistic Regression or SVM** if balanced performance across all sentiments is desired.
2. Consider **class weighting or resampling** (e.g., SMOTE) for tree-based models to improve Negative/Positive detection.
3. Focus on improving **Negative recall** → more labeled data, oversampling, or threshold tuning.
4. Always report **Macro F1** in addition to Accuracy, as it is more informative for imbalanced multiclass sentiment data.

Conclusions

Model Performance Conclusions

- **Logistic Regression** is the most suitable model for practical business use, even though it is not the highest in accuracy.
  - It achieved a **macro F1 score of 0.5723**, meaning it balances performance across **negative, neutral, and positive** classes better than the other models.
  - This balance is essential for understanding the full sentiment landscape around Apple and Google products.
- **Accuracy alone is misleading** in imbalanced datasets like this one.
  - For example, **XGBoost achieved 66.95% accuracy**, which seems high, **but it only had 10% recall for negative tweets** — meaning it failed to detect **9 out of 10 negative tweets**.
  - In real business scenarios such as **crisis management, customer support, or PR monitoring**, missing negative tweets is a **critical failure**, making XGBoost unsuitable despite its high accuracy.

Overall, the results show that **balanced performance (macro F1)** is more important than accuracy when analyzing public sentiment, especially when negative feedback is rare but highly important.

Business Insights

Sentiment Distribution Patterns

- **Neutral tweets dominate (60.3%)**, showing that most discussions around Apple and Google are:
  - Informational
  - News updates
  - General statements without emotional tone
- **Positive sentiment (33.3%)** is much higher than negative, suggesting:
  - A generally **favorable public perception** of both brands
  - Strong brand loyalty and satisfaction among users
- **Negative sentiment is very low (6.4%)**, but should be monitored, since shifts here can indicate early reputational or product issues.

Model Limitations Impact

- Detecting **negative sentiment remains challenging** — even the best model (SVM) reached only **54% recall** for negative tweets.
- This means:
  - Nearly **half of negative tweets go undetected**.
  - Businesses may **miss early signals of customer dissatisfaction**, complaints, or emerging crises.
  - Customer support teams may **fail to respond quickly** to real issues.
  - Negative sentiment spikes might go unnoticed until the problem becomes larger.

Implication:
For Apple and Google, failing to catch negative sentiment early can lead to **delayed crisis response, damaged brand reputation**, and **lost customer trust**.

Business Recommendations

1. Immediate Implementation Strategy

- **Use Logistic Regression or SVM** as the primary model due to their balanced performance across all sentiment classes.
- **Set up sentiment alerts** to detect spikes in negative sentiment early, compensating for the 51–54% recall rate on negative tweets.

---

2. Data Collection & Model Improvement

- **Collect more negative sentiment data** to fix severe class imbalance (only 6.4% negative tweets).
- **Apply class balancing techniques** such as SMOTE or class weighting to improve minority class detection.
- **Enhance feature engineering** with emojis, punctuation, capitalization, and better embeddings to reduce confusion between Negative and Neutral tweets.

---

3. Marketing Strategy Applications

- **Compare Apple vs Google sentiment** directly to identify brand-specific perception differences.
- **Track sentiment during product launches** to measure campaign success or detect rising issues.
- **Extract high-impact keywords** to guide marketing language, ad copy, and content creation.

---

4. Customer Support Optimization

- **Prioritize negative tweets** using a sentiment-based routing system for faster response.
- **Use a two-stage system:** automatic model screening + human review for borderline cases to improve accuracy.

---

5. Product Development Intelligence

- **Analyze sentiment by product features** to pinpoint what users love or dislike.
- **Identify recurring negative themes** (bugs, complaints, frustrations) for roadmap and quality improvements.

---

6. Long-term Strategic Initiatives

- **Retrain models quarterly** to adapt to evolving social media language and trends.
- **Explore advanced NLP models** like BERT for improved contextual understanding.
- **Create brand-specific sentiment models** for Apple and Google to capture unique vocabulary and patterns.
- **Build real-time dashboards** showing sentiment trends, alerts, and activity for executives and marketing teams.

In [ ]: