Sentiment Analysis using Amazon Polarity Dataset

This notebook presents the development of a sentiment analysis system using the Amazon Polarity dataset. The objective is to classify customer reviews as either positive or negative using Natural Language Processing (NLP) and machine learning techniques.

The workflow of this notebook includes:

- Data sampling and exploratory data analysis (EDA)
- Text feature extraction using TF-IDF
- Training and evaluation of multiple machine learning models
- Hyperparameter tuning
- Result comparison and analysis

In [2]:
```
# Import necessary libraries

# Data handling libraries
import pandas as pd
import numpy as np

# Dataset loading from Hugging Face
from datasets import load_dataset

# Visualization libraries
import matplotlib.pyplot as plt
import seaborn as sns

# Machine learning utilities
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.feature_extraction.text import TfidfVectorizer

# Classification models
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import LinearSVC

# Evaluation metrics
from sklearn.metrics import accuracy_score, precision_score, recall_score, f
from sklearn.metrics import confusion_matrix, classification_report, Confusi
```

Dataset Loading

The Amazon Polarity dataset contains product reviews labeled as:

- 0 → Negative sentiment
- 1 → Positive sentiment

Each data instance includes a review title, review content, and a sentiment label.

In [4]:
```
from datasets import load_dataset

# Load Amazon Polarity dataset from Hugging Face
dataset = load_dataset("amazon_polarity")
```

Dataset Combination

The Amazon Polarity dataset is provided with predefined splits. To ensure that sampling is representative of the entire dataset, all available splits are combined before further processing.

In [6]:
```
# Convert dataset splits to pandas DataFrames
train_df = pd.DataFrame(dataset["train"])
test_df = pd.DataFrame(dataset["test"])

# Combine all available splits into a single DataFrame
df = pd.concat([train_df, test_df], ignore_index=True)

# Display combined dataset size
print("Combined Dataset Shape:", df.shape)
```

```
Combined Dataset Shape: (4000000, 3)
```

Data Sampling Justification

The combined Amazon Polarity dataset contains 4 million records, which makes full-scale analysis computationally expensive.

A representative random sample is selected for exploratory data analysis and model development. This approach preserves the overall data distribution while enabling efficient experimentation.

In [8]:
```
# Randomly sample a manageable subset from the combined dataset
df = df.sample(n=50000, random_state=42)

# Combine title and review content into a single text feature
df["text"] = df["title"] + " " + df["content"]

# Display first 5 rows
df.head()
```

Out[8]:

	label	title	content	text
1049554	0	Deeply disappointing, faulty morality & social...	The book delves very little into art, aside fr...	Deeply disappointing, faulty morality & social...
214510	1	insight into the philosophy of libertarian soc...	In "The Limits of State Action" Enlightenment ...	insight into the philosophy of libertarian soc...
2145764	1	a great book	"In vain did the Bedouins strive to cut down a...	a great book "In vain did the Bedouins strive ...
2198867	0	toys for great sex	wow, that was bad, I threw it away after watch...	toys for great sex wow, that was bad, I threw ...
1184366	1	i love this movie!!!!	i just finished reading Someone Like You and i...	i love this movie!!!! i just finished reading ...

Exploratory Data Analysis (EDA)

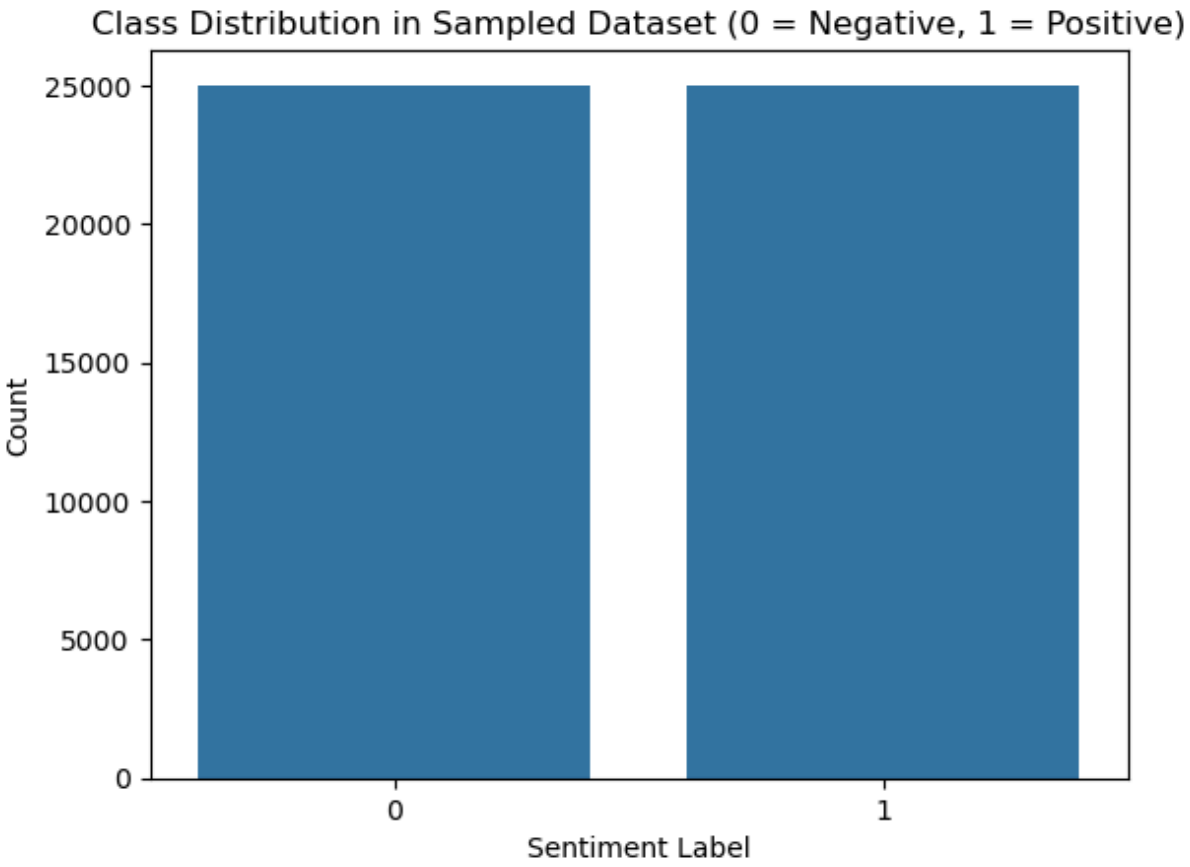
EDA is conducted on the sampled dataset to understand its structure, size, and sentiment class distribution.

In [10]:
```
# Display dataset shape
print("Dataset Shape:", df.shape)

# Display count of each sentiment label
df["label"].value_counts()
```

```
Dataset Shape: (50000, 4)
```

Out[10]:
```
label
0    25020
1    24980
Name: count, dtype: int64
```

In [11]:
```
# Visualize class distribution
sns.countplot(x="label", data=df)
plt.title("Class Distribution in Sampled Dataset (0 = Negative, 1 = Positive
plt.xlabel("Sentiment Label")
plt.ylabel("Count")
plt.show()
```



The class distribution indicates that the sampled dataset is balanced, with a similar number of positive and negative reviews. This balance helps ensure that the models are not biased toward any single class during training.

Data Exploration Visualizations

This section presents visualizations to better understand the sampled dataset, including sentiment class distribution and frequently occurring words in reviews.

Word Cloud for Positive Reviews

This word cloud shows the most frequent words appearing in positive reviews. Larger words indicate higher frequency.

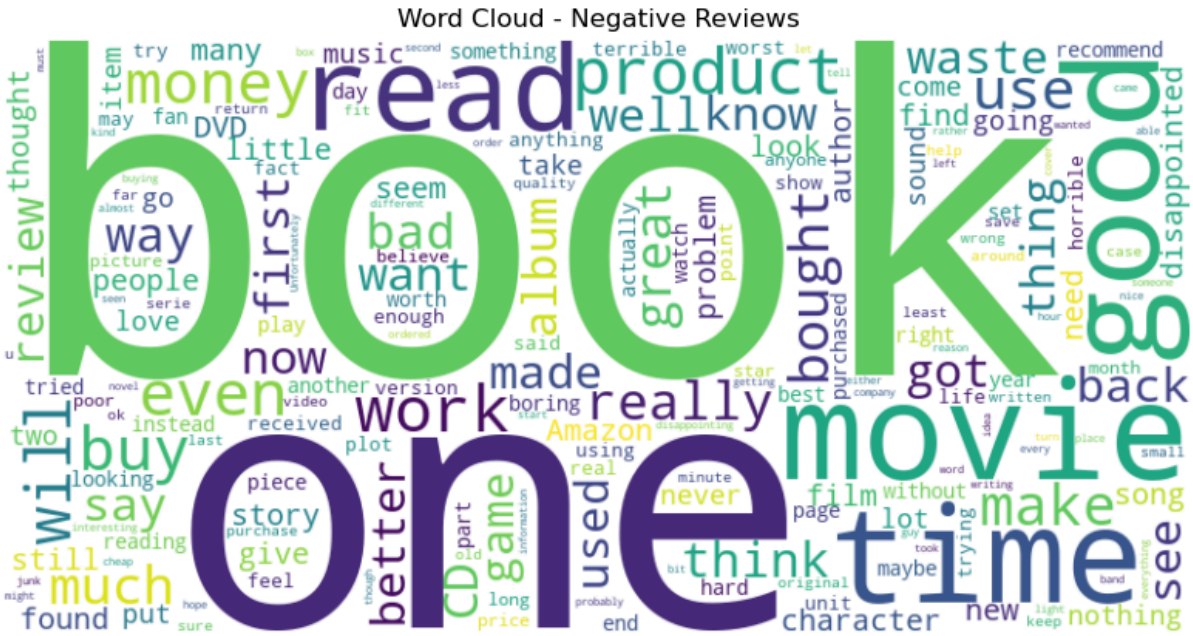
In [15]:
```
from wordcloud import WordCloud, STOPWORDS

# Combine all positive review text
positive_text = " ".join(df[df["label"] == 1]["text"])

# Generate word cloud
wordcloud_pos = WordCloud(
    width=800,
    height=400,
    background_color="white",
```

```
    stopwords=STOPWORDS
).generate(positive_text)

# Display word cloud
plt.figure(figsize=(10,5))
plt.imshow(wordcloud_pos.to_image(), interpolation="bilinear")
plt.axis("off")
plt.title("Word Cloud - Positive Reviews")
plt.show()
```

Word Cloud for Negative Reviews

This word cloud highlights commonly used words in negative customer reviews.

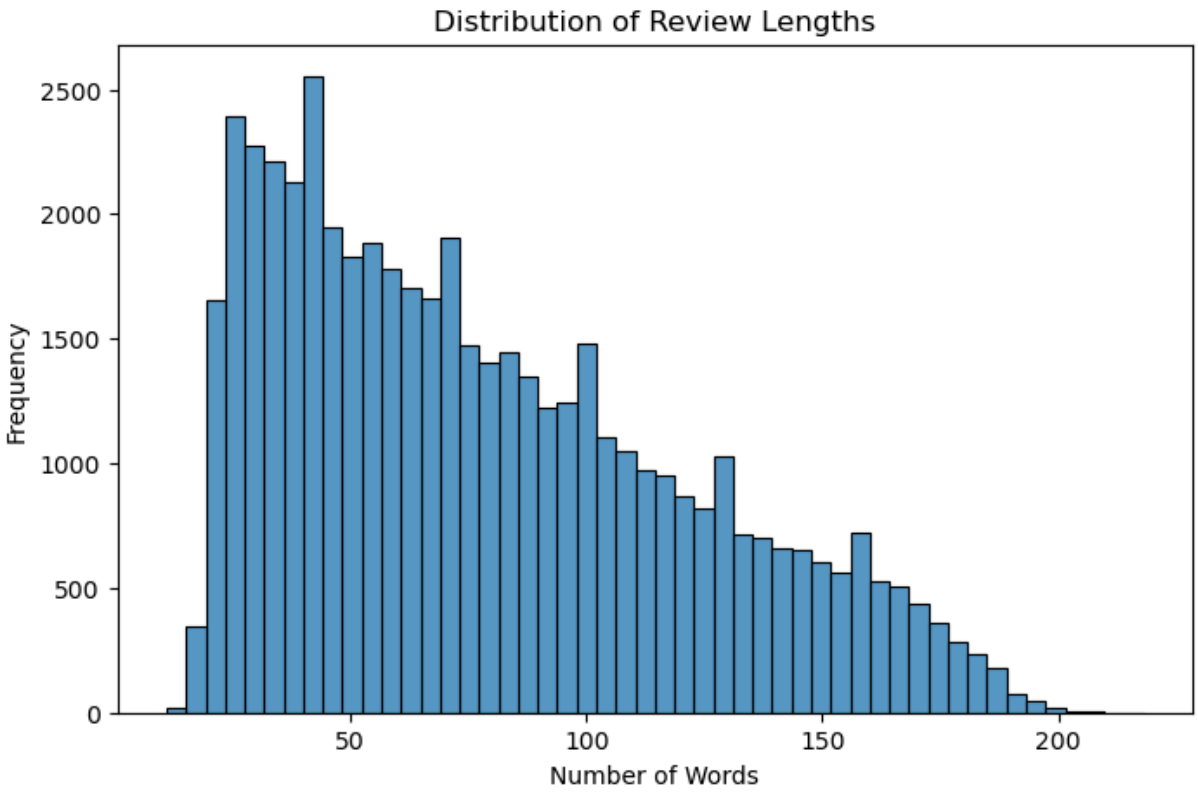
In [17]:
```
# Combine all negative review text
negative_text = " ".join(df[df["label"] == 0]["text"])

# Generate word cloud
wordcloud_neg = WordCloud(
    width=800,
    height=400,
    background_color="white",
    stopwords=STOPWORDS
).generate(negative_text)

# Display word cloud
plt.figure(figsize=(10,5))
plt.imshow(wordcloud_neg.to_image(), interpolation="bilinear")
plt.axis("off")
plt.title("Word Cloud - Negative Reviews")
plt.show()
```



Review Length Distribution

This visualization shows the distribution of review lengths in the dataset.

In [19]:
```
# Calculate review length
df["review_length"] = df["text"].apply(lambda x: len(x.split()))

# Plot distribution
plt.figure(figsize=(8,5))
sns.histplot(df["review_length"], bins=50)
plt.title("Distribution of Review Lengths")
plt.xlabel("Number of Words")
plt.ylabel("Frequency")
plt.show()
```



The class distribution visualization confirms that the dataset is balanced. Word clouds reveal meaningful and sentiment-related terms commonly used in positive and negative reviews. The review length distribution shows variability in customer feedback, which supports the need for text normalization and feature extraction.

Train–Test Split

The dataset is divided into training and testing sets using an 80–20 split. This allows the models to be evaluated on unseen data.

In [22]:
```
# Define features and target
X = df["text"]
y = df["label"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42,
    stratify=y  # preserves class balance
)
```

Feature Extraction using TF-IDF

TF-IDF (Term Frequency–Inverse Document Frequency) converts text data into numerical features while reducing the influence of commonly occurring words.

In [24]:
```python
# Initialize TF-IDF vectorizer
tfidf = TfidfVectorizer(
    stop_words="english",
    max_features=15000
)

# Fit on training data and transform both sets
X_train_tfidf = tfidf.fit_transform(X_train)
X_test_tfidf = tfidf.transform(X_test)
```

Model Training and Evaluation

This section trains three machine learning models for sentiment analysis. Each model is evaluated before and after hyperparameter tuning.

Logistic Regression

In [27]:
```python
# Create Logistic Regression model with default parameters
lr = LogisticRegression(max_iter=1000)

# Train the model using TF-IDF transformed training data
lr.fit(X_train_tfidf, y_train)

# Predict sentiment labels for TF-IDF transformed test data
lr_pred = lr.predict(X_test_tfidf)

# Display classification report
print("Logistic Regression Classification Report")
print(classification_report(y_test, lr_pred))

# Calculate and store accuracy
lr_accuracy_before = accuracy_score(y_test, lr_pred)
```

```
Logistic Regression Classification Report
              precision    recall  f1-score   support

           0       0.88      0.88      0.88      5004
           1       0.88      0.87      0.88      4996

    accuracy                           0.88     10000
   macro avg       0.88      0.88      0.88     10000
weighted avg       0.88      0.88      0.88     10000
```

Sample Prediction Using Logistic Regression

In [29]:
```python
# Example reviews for prediction
samples = [
    "This product is excellent and exceeded my expectations",
    "Very poor quality and a complete waste of money"
```

```
]

# Transform samples and predict sentiment
sample_vec = tfidf.transform(samples)
sample_preds = lr.predict(sample_vec)

# Display predictions
for text, pred in zip(samples, sample_preds):
    sentiment = "Positive" if pred == 1 else "Negative"
    print(text)
    print("Predicted Sentiment:", sentiment)
    print()
```

```
This product is excellent and exceeded my expectations
Predicted Sentiment: Positive

Very poor quality and a complete waste of money
Predicted Sentiment: Negative
```

Logistic Regression – After Hyperparameter Tuning

The regularization parameter `C` is tuned using GridSearchCV to improve accuracy.

In [31]:
```
# Define hyperparameter grid
lr_param_grid = {
    "C": [0.01, 0.1, 1, 10]
}

# Apply GridSearchCV
lr_grid = GridSearchCV(
    LogisticRegression(max_iter=1000),
    lr_param_grid,
    cv=3,
    scoring="accuracy"
)

# Train using TF-IDF features
lr_grid.fit(X_train_tfidf, y_train)

# Get best model
best_lr = lr_grid.best_estimator_

# Predict using tuned model
lr_pred_after = best_lr.predict(X_test_tfidf)

# Display classification report
print("Logistic Regression Classification Report (After Tuning)")
print(classification_report(y_test, lr_pred_after))

# Accuracy after tuning
lr_accuracy_after = accuracy_score(y_test, lr_pred_after)
```

```
Logistic Regression Classification Report (After Tuning)
              precision    recall  f1-score   support

           0       0.88      0.88      0.88      5004
           1       0.88      0.87      0.88      4996

    accuracy                           0.88     10000
   macro avg       0.88      0.88      0.88     10000
weighted avg       0.88      0.88      0.88     10000
```

Sample Prediction Using Logistic Regression (after hyperparameter tuning)

In [33]:
```python
# Example reviews for prediction
samples = [
    "This product is excellent and exceeded my expectations",
    "Very poor quality and a complete waste of money"
]

# Transform samples and predict sentiment
sample_vec = tfidf.transform(samples)
sample_preds = best_lr.predict(sample_vec)

# Display predictions
for text, pred in zip(samples, sample_preds):
    sentiment = "Positive" if pred == 1 else "Negative"
    print(text)
    print("Predicted Sentiment:", sentiment)
    print()
```

```
This product is excellent and exceeded my expectations
Predicted Sentiment: Positive

Very poor quality and a complete waste of money
Predicted Sentiment: Negative
```

Multinomial Naive Bayes

In [35]:
```python
# Create Naive Bayes model
nb = MultinomialNB()

# Train model using TF-IDF features
nb.fit(X_train_tfidf, y_train)

# Predict on test data
nb_pred = nb.predict(X_test_tfidf)

# Display classification report
print("Naive Bayes Classification Report")
print(classification_report(y_test, nb_pred))
```

```
# Accuracy before tuning
nb_accuracy_before = accuracy_score(y_test, nb_pred)
```

```
Naive Bayes Classification Report
              precision    recall  f1-score   support

           0       0.83      0.84      0.84      5004
           1       0.84      0.83      0.84      4996

    accuracy                           0.84     10000
   macro avg       0.84      0.84      0.84     10000
weighted avg       0.84      0.84      0.84     10000
```

Sample Prediction Using Multinomial Naive Bayes

In [37]:
```
# Example reviews for prediction
samples = [
    "This product is excellent and exceeded my expectations",
    "Very poor quality and a complete waste of money"
]

# Transform samples and predict sentiment
sample_vec = tfidf.transform(samples)
sample_preds = nb.predict(sample_vec)

# Display predictions
for text, pred in zip(samples, sample_preds):
    sentiment = "Positive" if pred == 1 else "Negative"
    print(text)
    print("Predicted Sentiment:", sentiment)
    print()
```

```
This product is excellent and exceeded my expectations
Predicted Sentiment: Positive

Very poor quality and a complete waste of money
Predicted Sentiment: Negative
```

Multinomial Naive Bayes (after Hyperparameter Tuning)

In [39]:
```
# Define hyperparameter grid
nb_param_grid = {
    "alpha": [0.1, 0.5, 1.0]
}

# Apply GridSearchCV
nb_grid = GridSearchCV(
    MultinomialNB(),
    nb_param_grid,
    cv=3,
    scoring="accuracy"
```

```
)

# Train using TF-IDF features
nb_grid.fit(X_train_tfidf, y_train)

# Get best model
best_nb = nb_grid.best_estimator_

# Predict using tuned model
nb_pred_after = best_nb.predict(X_test_tfidf)

# Display classification report
print("Naive Bayes Classification Report (After Tuning)")
print(classification_report(y_test, nb_pred_after))

# Accuracy after tuning
nb_accuracy_after = accuracy_score(y_test, nb_pred_after)
```

```
Naive Bayes Classification Report (After Tuning)
              precision    recall  f1-score   support

           0       0.83      0.84      0.84      5004
           1       0.84      0.83      0.84      4996

    accuracy                           0.84     10000
   macro avg       0.84      0.84      0.84     10000
weighted avg       0.84      0.84      0.84     10000
```

Sample Prediction Using Logistic Regression (after hyperparameter tuning)

In [41]:
```
# Example reviews for prediction
samples = [
    "This product is excellent and exceeded my expectations",
    "Very poor quality and a complete waste of money"
]

# Transform samples and predict sentiment
sample_vec = tfidf.transform(samples)
sample_preds = best_nb.predict(sample_vec)

# Display predictions
for text, pred in zip(samples, sample_preds):
    sentiment = "Positive" if pred == 1 else "Negative"
    print(text)
    print("Predicted Sentiment:", sentiment)
    print()
```

```
This product is excellent and exceeded my expectations
Predicted Sentiment: Positive

Very poor quality and a complete waste of money
Predicted Sentiment: Negative
```

Linear SVM

In [43]:
```python
# Create Linear SVM model
svm = LinearSVC()

# Train model using TF-IDF features
svm.fit(X_train_tfidf, y_train)

# Predict on test data
svm_pred = svm.predict(X_test_tfidf)

# Display classification report
print("Linear SVM Classification Report")
print(classification_report(y_test, svm_pred))

# Accuracy before tuning
svm_accuracy_before = accuracy_score(y_test, svm_pred)
```

```
Linear SVM Classification Report
              precision    recall  f1-score   support

           0       0.86      0.87      0.87      5004
           1       0.87      0.86      0.86      4996

    accuracy                           0.86     10000
   macro avg       0.86      0.86      0.86     10000
weighted avg       0.86      0.86      0.86     10000
```

Sample Prediction Using Linear SVM

In [45]:
```python
# Example reviews for prediction
samples = [
    "This product is excellent and exceeded my expectations",
    "Very poor quality and a complete waste of money"
]

# Transform samples and predict sentiment
sample_vec = tfidf.transform(samples)
sample_preds = svm.predict(sample_vec)

# Display predictions
for text, pred in zip(samples, sample_preds):
    sentiment = "Positive" if pred == 1 else "Negative"
    print(text)
    print("Predicted Sentiment:", sentiment)
    print()
```

```
This product is excellent and exceeded my expectations
Predicted Sentiment: Positive

Very poor quality and a complete waste of money
Predicted Sentiment: Negative
```

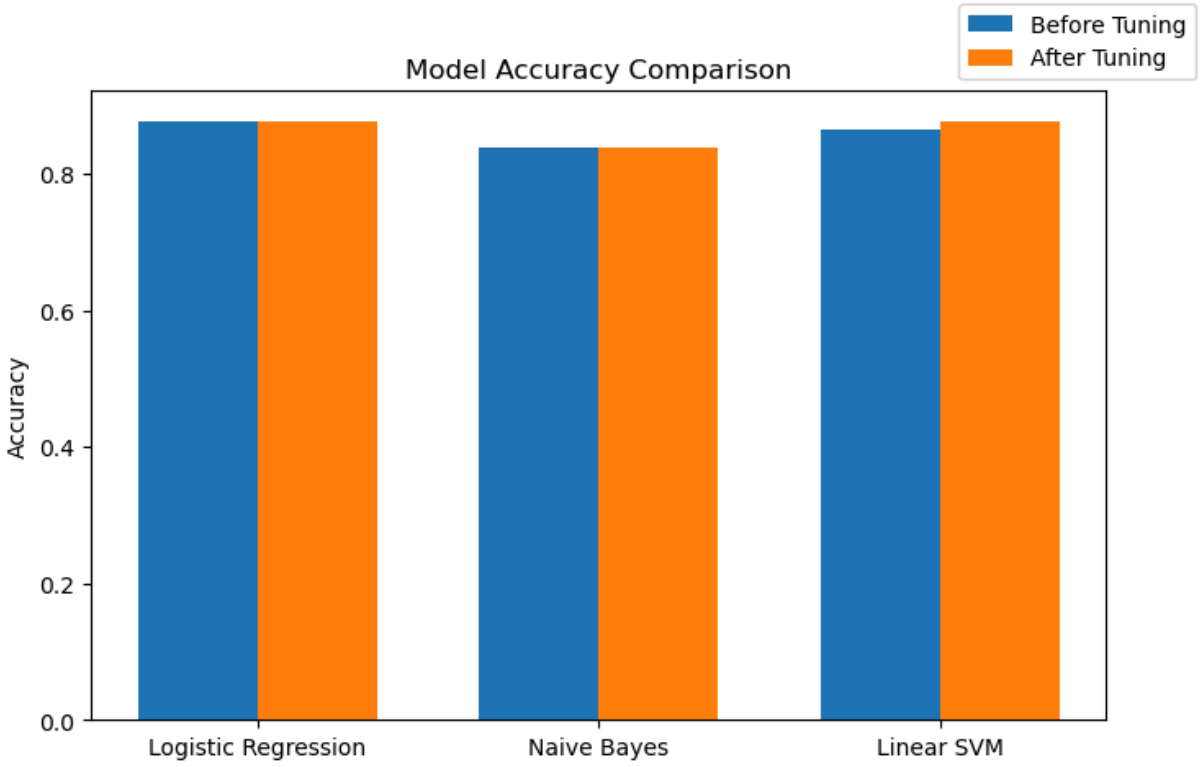
Linear SVM (after Hyperparameter Tuning)

In [47]:
```python
# Define hyperparameter grid
svm_param_grid = {
    "C": [0.01, 0.1, 1, 10]
}

# Apply GridSearchCV
svm_grid = GridSearchCV(
    LinearSVC(),
    svm_param_grid,
    cv=3,
    scoring="accuracy"
)

# Train using TF-IDF features
svm_grid.fit(X_train_tfidf, y_train)

# Get best model
best_svm = svm_grid.best_estimator_

# Predict using tuned model
svm_pred_after = best_svm.predict(X_test_tfidf)

# Display classification report
print("Linear SVM Classification Report (After Tuning)")
print(classification_report(y_test, svm_pred_after))

# Accuracy after tuning
svm_accuracy_after = accuracy_score(y_test, svm_pred_after)
```

```
Linear SVM Classification Report (After Tuning)
              precision    recall  f1-score   support

           0       0.88      0.88      0.88      5004
           1       0.88      0.88      0.88      4996

    accuracy                           0.88     10000
   macro avg       0.88      0.88      0.88     10000
weighted avg       0.88      0.88      0.88     10000
```

Accuracy Comparison Before and After Hyperparameter Tuning

In [49]:
```python
# Store model names
models = ["Logistic Regression", "Naive Bayes", "Linear SVM"]

# Store accuracy values
before = [lr_accuracy_before, nb_accuracy_before, svm_accuracy_before]
after = [lr_accuracy_after, nb_accuracy_after, svm_accuracy_after]

# Create bar positions
x = np.arange(len(models))
width = 0.35

# Plot bar graph
plt.figure(figsize=(8,5))
plt.bar(x - width/2, before, width, label="Before Tuning")
plt.bar(x + width/2, after, width, label="After Tuning")

# Label axes and title
plt.xticks(x, models)
plt.ylabel("Accuracy")
plt.title("Model Accuracy Comparison")
plt.legend(bbox_to_anchor=(1.1, 1), loc='lower right')

# Display plot
plt.show()
```



ROC Curve Analysis

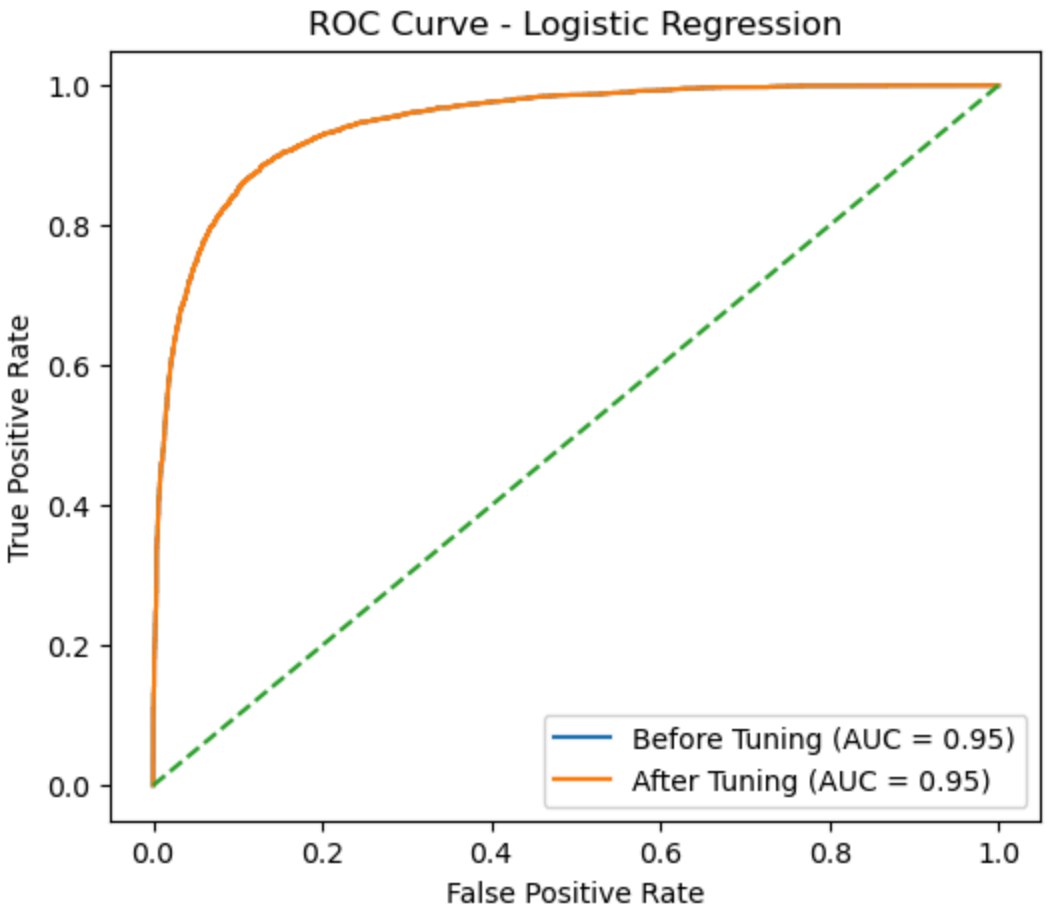
The Receiver Operating Characteristic (ROC) curve is used to evaluate the classification performance of models by comparing the True Positive Rate (TPR) and False Positive

Rate (FPR).

ROC curves are plotted for each model before and after hyperparameter tuning to visually assess improvement.

ROC Curve – Logistic Regression

In [52]:
```
from sklearn.metrics import roc_curve, auc

# Get probability scores BEFORE tuning
lr_probs_before = lr.predict_proba(X_test_tfidf)[:, 1]

# Get probability scores AFTER tuning
lr_probs_after = best_lr.predict_proba(X_test_tfidf)[:, 1]

# Compute ROC values before tuning
lr_fpr_before, lr_tpr_before, _ = roc_curve(y_test, lr_probs_before)
lr_auc_before = auc(lr_fpr_before, lr_tpr_before)

# Compute ROC values after tuning
lr_fpr_after, lr_tpr_after, _ = roc_curve(y_test, lr_probs_after)
lr_auc_after = auc(lr_fpr_after, lr_tpr_after)

# Plot ROC curve
plt.figure(figsize=(6,5))
plt.plot(lr_fpr_before, lr_tpr_before, label=f"Before Tuning (AUC = {lr_auc_
plt.plot(lr_fpr_after, lr_tpr_after, label=f"After Tuning (AUC = {lr_auc_aft
plt.plot([0, 1], [0, 1], linestyle="--")

plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve - Logistic Regression")
plt.legend()
plt.show()
```

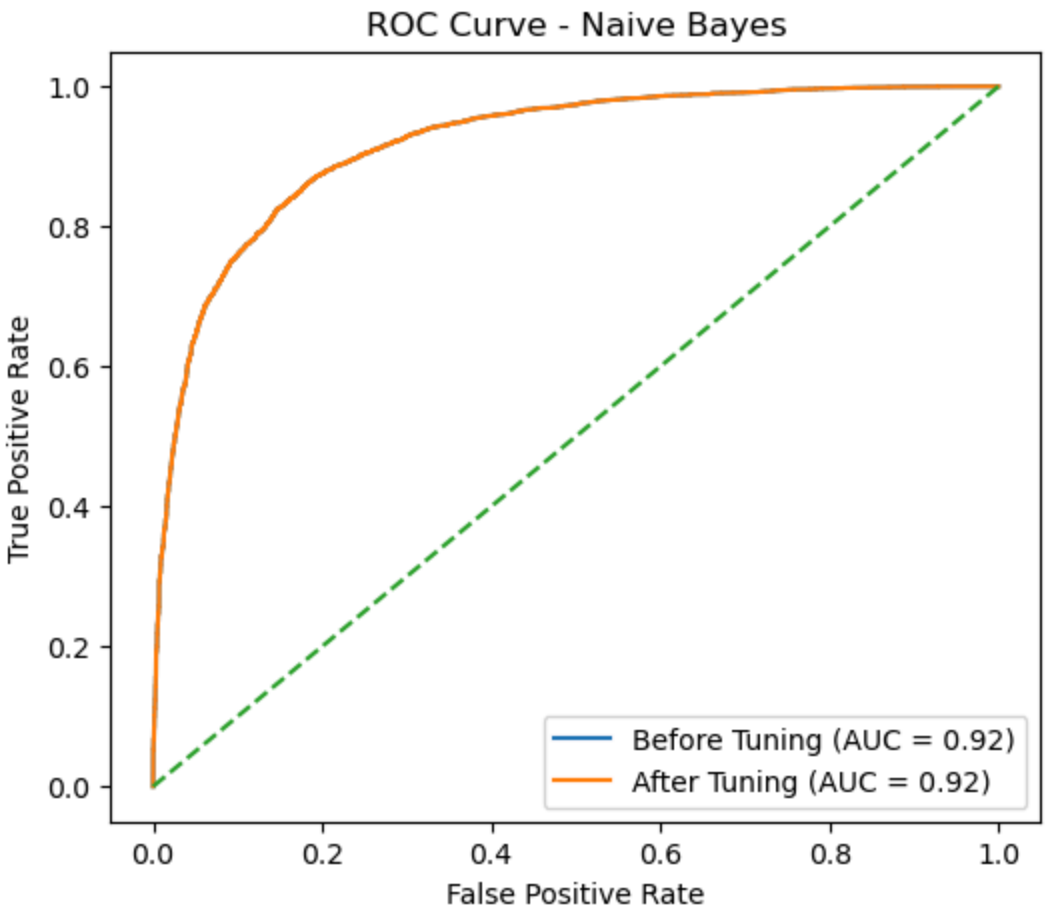


ROC Curve – Multinomial Naive Bayes

In [54]:

```
# Get probability scores BEFORE tuning
nb_probs_before = nb.predict_proba(X_test_tfidf)[:, 1]

# Get probability scores AFTER tuning
nb_probs_after = best_nb.predict_proba(X_test_tfidf)[:, 1]

# Compute ROC values before tuning
nb_fpr_before, nb_tpr_before, _ = roc_curve(y_test, nb_probs_before)
nb_auc_before = auc(nb_fpr_before, nb_tpr_before)

# Compute ROC values after tuning
nb_fpr_after, nb_tpr_after, _ = roc_curve(y_test, nb_probs_after)
nb_auc_after = auc(nb_fpr_after, nb_tpr_after)

# Plot ROC curve
plt.figure(figsize=(6,5))
plt.plot(nb_fpr_before, nb_tpr_before, label=f"Before Tuning (AUC = {nb_auc_
plt.plot(nb_fpr_after, nb_tpr_after, label=f"After Tuning (AUC = {nb_auc_aft
plt.plot([0, 1], [0, 1], linestyle="--")

plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve - Naive Bayes")
```

```
plt.legend()
plt.show()
```

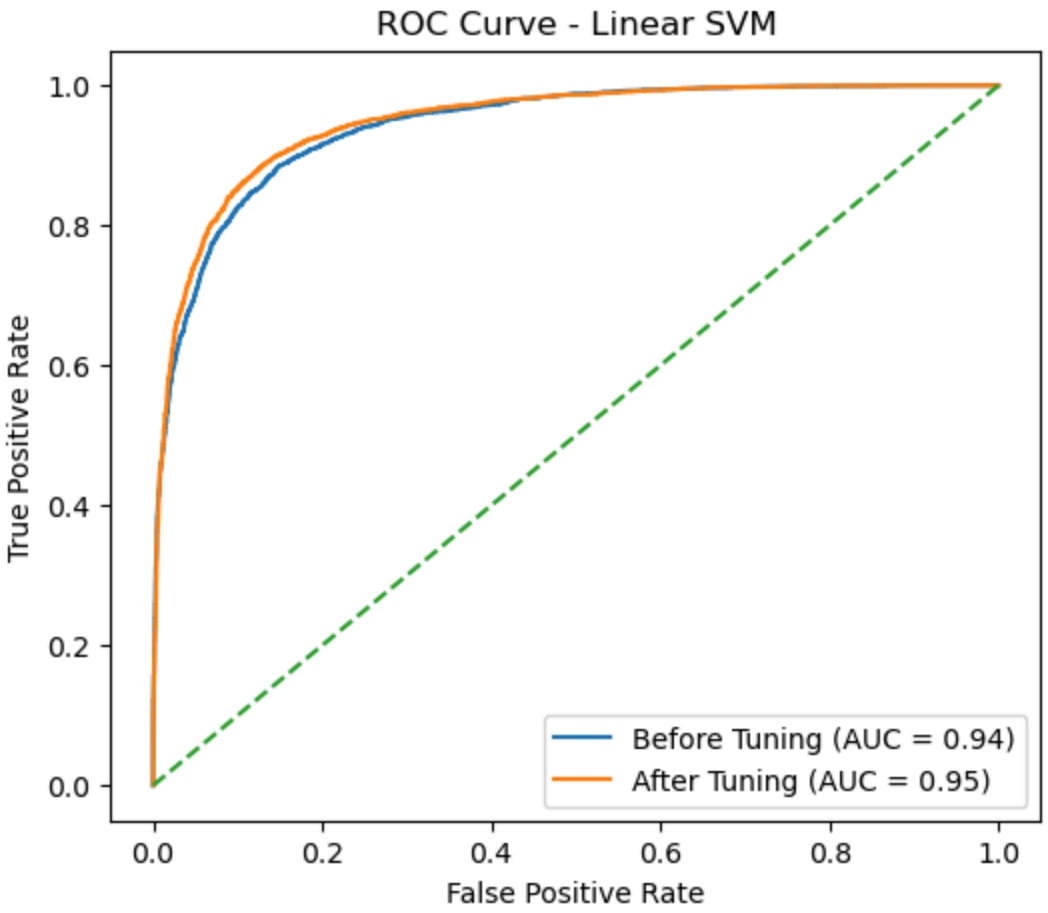


ROC Curve – Linear SVM

Linear SVM does not provide probability estimates directly. Therefore, the decision function output is used instead to compute ROC curves.

In [56]:
```
# Get decision scores BEFORE tuning
svm_scores_before = svm.decision_function(X_test_tfidf)

# Get decision scores AFTER tuning
svm_scores_after = best_svm.decision_function(X_test_tfidf)

# Compute ROC values before tuning
svm_fpr_before, svm_tpr_before, _ = roc_curve(y_test, svm_scores_before)
svm_auc_before = auc(svm_fpr_before, svm_tpr_before)

# Compute ROC values after tuning
svm_fpr_after, svm_tpr_after, _ = roc_curve(y_test, svm_scores_after)
svm_auc_after = auc(svm_fpr_after, svm_tpr_after)

# Plot ROC curve
plt.figure(figsize=(6,5))
plt.plot(svm_fpr_before, svm_tpr_before, label=f"Before Tuning (AUC = {svm_a
plt.plot(svm_fpr_after, svm_tpr_after, label=f"After Tuning (AUC = {svm_auc_
```

```
plt.plot([0, 1], [0, 1], linestyle="--")

plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve - Linear SVM")
plt.legend()
plt.show()
```

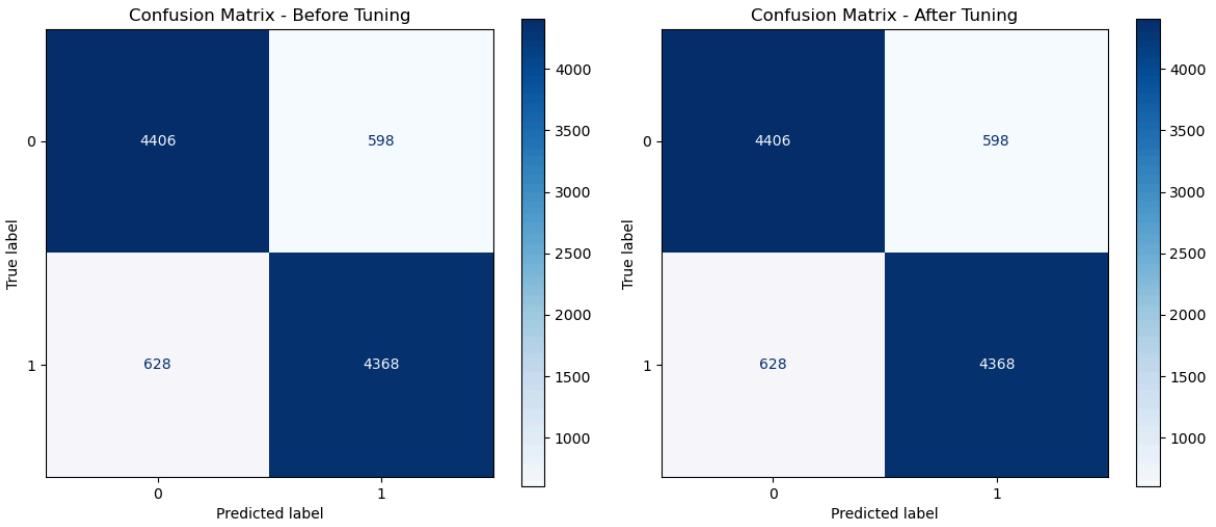


Confusion Matrix

Confusion Matrix - Linear Regression

In [59]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Get predictions BEFORE tuning
lr_preds_before = lr.predict(X_test_tfidf)

# Get predictions AFTER tuning
lr_preds_after = best_lr.predict(X_test_tfidf)

# Create confusion matrices
cm_before = confusion_matrix(y_test, lr_preds_before)
cm_after = confusion_matrix(y_test, lr_preds_after)
```

```python
# Plot confusion matrices side by side
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Before tuning
disp_before = ConfusionMatrixDisplay(confusion_matrix=cm_before)
disp_before.plot(ax=axes[0], cmap='Blues', values_format='d')
axes[0].set_title('Confusion Matrix - Before Tuning')

# After tuning
disp_after = ConfusionMatrixDisplay(confusion_matrix=cm_after)
disp_after.plot(ax=axes[1], cmap='Blues', values_format='d')
axes[1].set_title('Confusion Matrix - After Tuning')

plt.tight_layout()
plt.show()

# Print detailed metrics
print("BEFORE TUNING:")
print(f"True Negatives: {cm_before[0,0]}")
print(f"False Positives: {cm_before[0,1]}")
print(f"False Negatives: {cm_before[1,0]}")
print(f"True Positives: {cm_before[1,1]}")
print()
print("AFTER TUNING:")
print(f"True Negatives: {cm_after[0,0]}")
print(f"False Positives: {cm_after[0,1]}")
print(f"False Negatives: {cm_after[1,0]}")
print(f"True Positives: {cm_after[1,1]}")
```

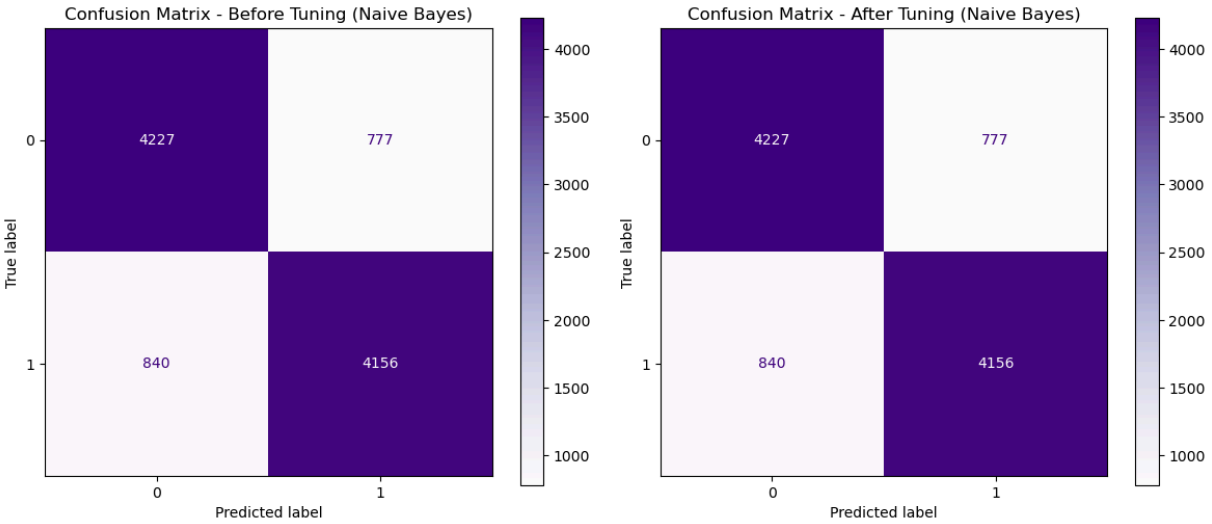



```
BEFORE TUNING:
True Negatives: 4406
False Positives: 598
False Negatives: 628
True Positives: 4368

AFTER TUNING:
True Negatives: 4406
False Positives: 598
False Negatives: 628
True Positives: 4368
```

Confusion Matrix - Naive Bayes

In [61]:

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Get predictions BEFORE tuning
nb_preds_before = nb.predict(X_test_tfidf)

# Get predictions AFTER tuning
nb_preds_after = best_nb.predict(X_test_tfidf)

# Create confusion matrices
cm_before = confusion_matrix(y_test, nb_preds_before)
cm_after = confusion_matrix(y_test, nb_preds_after)

# Plot confusion matrices side by side
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Before tuning
disp_before = ConfusionMatrixDisplay(confusion_matrix=cm_before)
disp_before.plot(ax=axes[0], cmap='Purples', values_format='d')
axes[0].set_title('Confusion Matrix - Before Tuning (Naive Bayes)')

# After tuning
disp_after = ConfusionMatrixDisplay(confusion_matrix=cm_after)
disp_after.plot(ax=axes[1], cmap='Purples', values_format='d')
axes[1].set_title('Confusion Matrix - After Tuning (Naive Bayes)')

plt.tight_layout()
plt.show()

# Print detailed metrics
print("NAIVE BAYES - BEFORE TUNING:")
print(f"True Negatives: {cm_before[0,0]}")
print(f"False Positives: {cm_before[0,1]}")
print(f"False Negatives: {cm_before[1,0]}")
print(f"True Positives: {cm_before[1,1]}")
print()
print("NAIVE BAYES - AFTER TUNING:")
print(f"True Negatives: {cm_after[0,0]}")
print(f"False Positives: {cm_after[0,1]}")
print(f"False Negatives: {cm_after[1,0]}")
print(f"True Positives: {cm_after[1,1]}")
```



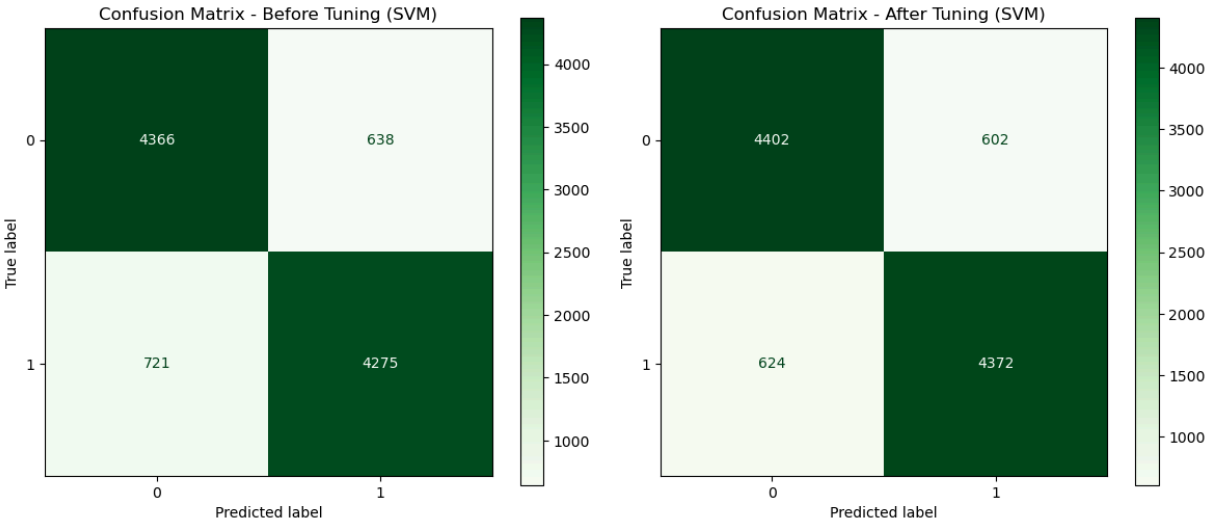

```
NAIVE BAYES - BEFORE TUNING:
True Negatives: 4227
False Positives: 777
False Negatives: 840
True Positives: 4156

NAIVE BAYES - AFTER TUNING:
True Negatives: 4227
False Positives: 777
False Negatives: 840
True Positives: 4156
```

Confusion Matrix - Linear SVM

In [63]:

```python
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Get predictions BEFORE tuning
svm_preds_before = svm.predict(X_test_tfidf)

# Get predictions AFTER tuning
svm_preds_after = best_svm.predict(X_test_tfidf)

# Create confusion matrices
cm_before = confusion_matrix(y_test, svm_preds_before)
cm_after = confusion_matrix(y_test, svm_preds_after)

# Plot confusion matrices side by side
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Before tuning
disp_before = ConfusionMatrixDisplay(confusion_matrix=cm_before)
disp_before.plot(ax=axes[0], cmap='Greens', values_format='d')
axes[0].set_title('Confusion Matrix - Before Tuning (SVM)')

# After tuning
disp_after = ConfusionMatrixDisplay(confusion_matrix=cm_after)
disp_after.plot(ax=axes[1], cmap='Greens', values_format='d')
axes[1].set_title('Confusion Matrix - After Tuning (SVM)')
```

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

# Print detailed metrics
print("SVM - BEFORE TUNING:")
print(f"True Negatives: {cm_before[0,0]}")
print(f"False Positives: {cm_before[0,1]}")
print(f"False Negatives: {cm_before[1,0]}")
print(f"True Positives: {cm_before[1,1]}")
print()
print("SVM - AFTER TUNING:")
print(f"True Negatives: {cm_after[0,0]}")
print(f"False Positives: {cm_after[0,1]}")
print(f"False Negatives: {cm_after[1,0]}")
print(f"True Positives: {cm_after[1,1]}")
```




```
SVM - BEFORE TUNING:
True Negatives: 4366
False Positives: 638
False Negatives: 721
True Positives: 4275

SVM - AFTER TUNING:
True Negatives: 4402
False Positives: 602
False Negatives: 624
True Positives: 4372
```

Final Model Performance Summary

In [65]:

```
# Create summary table for comparison
summary_df = pd.DataFrame({
    "Model": models,
    "Accuracy Before Tuning": before,
    "Accuracy After Tuning": after
})
```

```
# Display summary table
summary_df
```

Out[65]:

	Model	Accuracy Before Tuning	Accuracy After Tuning
0	Logistic Regression	0.8774	0.8774
1	Naive Bayes	0.8383	0.8383
2	Linear SVM	0.8641	0.8774