

Note: The CSV dataset provided with Lecture 9 was used as the dataset for this classification project.

```
In [ ]: # CELL 1
# Note: Import the necessary libraries for data processing, feature representation,
# machine learning models, evaluation, and visualization.

import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split, GridSearchCV

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import classification_report

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
In [ ]: # CELL 2
# Note: Mount Google Drive and read the NLP dataset directly from Google Drive.

from google.colab import drive

# Mount Google Drive
drive.mount('/content/drive')

# Read the dataset from Google Drive
df = pd.read_csv(
    "/content/drive/MyDrive/My Project 2/unbalanceddataset (1).csv",
    encoding="ISO-8859-1"
)

df.head()
```

Mounted at /content/drive

Out[]:

	text	sentiment
0	Java Concurrency in Practice is probably the b...	positive
1	haha aww hun i bet you are more creative tha...	positive
2	_pickle lol, thank you very much Hope you're h...	positive
3	Out for an evening on the town with jeremy. Sa...	negative
4	- just took over the #1 Most Endorsed spot on...	positive

```
In [ ]: # CELL 3
# Note: Check the number of rows in the dataset.

len(df)
```

Out[]: 2600

```
In [ ]: # CELL 4
# Note: Check the number of samples in each sentiment class.
# This confirms the dataset is imbalanced (positive >> negative).

df['sentiment'].value_counts()
```

Out[]:

	count
sentiment	
positive	2000
negative	600

dtype: int64

```
In [ ]: # CELL 5
# Note: Import the libraries required for NLP text preprocessing.

import re
import spacy

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

import nltk
```

```
In [ ]: # CELL 6
# Note: Download the NLTK resources required for tokenization and stop-word removal.

nltk.download('punkt')
nltk.download('punkt_tab')
nltk.download('stopwords')
nltk.download('wordnet')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package wordnet to /root/nltk_data...
```

Out[]: True

```
In [ ]: # CELL 7
# Note: Create the Porter Stemmer and Load the English stop words List.

stemmer = PorterStemmer()
```

```
# Stop words List
stop_words = set(stopwords.words('english'))
```

```
In [ ]: # CELL 8
# Note: Define a preprocessing function that cleans the text step by step.
# The steps include Lowercasing, removing URLs, removing HTML tags,
# removing special characters, tokenization, stop-word removal, and stemming.
# This follows the exact preprocessing pipeline covered in the NLP lecture.

def preprocess_text(text):

    # 1. Lowercasing
    text = text.lower()

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

    # 3. Remove HTML tags
    text = re.sub(r'<.*?>', '', text)

    # 4. Remove Special Characters, Numbers, and Punctuation
    text = re.sub(r'^a-zA-Z\s', '', text)

    # 5. Tokenization
    tokens = word_tokenize(text)

    # 6. Remove Stop Words
    tokens = [word for word in tokens if word not in stop_words]

    # 7. Stemming
    stemmed_tokens = [stemmer.stem(word) for word in tokens]

    # 8. Final clean text reconstruction (optional)
    clean_text = ' '.join(stemmed_tokens)

    return clean_text
```

```
In [ ]: # CELL 9
# Note: Drop any rows with missing text before preprocessing (safety check),
# then apply the preprocessing function to create the clean_text column.

df = df.dropna(subset=['text'])

df["clean_text"] = df["text"].apply(preprocess_text)

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

```
Out[ ]:          clean_text
0  java concurr practic probabl best java book iv...
1          haha aww hun bet creativ
2    pickl lol thank much hope your great day
3    even town jeremi sad carri cant come
4    took endors spot twindexxcom thank endors

dtype: object
```

```
In [ ]: # CELL 10
# Note: Define the input features X and the target variable y.
# X contains the cleaned text and y contains the sentiment labels.

X = df['clean_text'].values

y = df['sentiment'].values
```

```
In [ ]: # CELL 11
# Note: Split the dataset into training and testing sets.
# 70% of the data is used for training and 30% for testing.
# Stratify is used to preserve the class distribution (important for an
# imbalanced dataset).

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.30,
    random_state=42,
    stratify=y
)
```

```
In [ ]: # CELL 12
# Note: Check the number of samples in the training and testing sets.

print("Training samples:", len(X_train))
print("Testing samples:", len(X_test))

Training samples: 1820
Testing samples: 780
```

```
In [ ]: # CELL 13
# Note: Apply Bag of Words feature representation.
# CountVectorizer converts the cleaned text into numerical word-frequency vectors.

vectorizer = CountVectorizer()

X_train_BOW = vectorizer.fit_transform(X_train)

X_test_BOW = vectorizer.transform(X_test)
```

```
In [ ]: # CELL 14
# Note: Check the dimensions of the Bag of Words training and testing matrices.

print("X_train_BOW shape:", X_train_BOW.shape)

print("X_test_BOW shape:", X_test_BOW.shape)

X_train_BOW shape: (1820, 3657)
X_test_BOW shape: (780, 3657)
```

```
In [ ]: # CELL 15
# Note: Define a function to evaluate the classification models
# using Accuracy, Precision, Recall, F1 Score, and Classification Report.
# A confusion matrix is also displayed. This matches the evaluate_model
# function used throughout the Lectures.

def evaluate_model(model_name, y_true, y_pred):

    # Calculate metrics
    accuracy = accuracy_score(y_true, y_pred)

    precision = precision_score(
```

```

        y_true,
        y_pred,
        average='weighted'
    )

    recall = recall_score(
        y_true,
        y_pred,
        average='weighted'
    )

    f1 = f1_score(
        y_true,
        y_pred,
        average='weighted'
    )

    cm = confusion_matrix(y_true, y_pred)

    # Classification report
    report = classification_report(y_true, y_pred)

    # Output results
    metrics = {
        'Model Name': model_name,
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'F1 Score': f1,
        'Classification Report': report
    }

    # Plot Confusion Matrix
    plt.figure(figsize=(5, 4))

    sns.heatmap(
        cm,
        annot=True,
        fmt='d',
        cmap='Blues',
        cbar=False,
        xticklabels=np.unique(y_true),
        yticklabels=np.unique(y_true)
    )

    plt.title(f'Confusion Matrix for {model_name}')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()

    return metrics

```

```

In [ ]: # CELL 16
# Note: Define, train, and evaluate the Random Forest Classifier
# using the Bag of Words feature representation.

# Define the Random Forest Classifier
rf_classifier = RandomForestClassifier(random_state=42)

# Define the hyperparameters for grid search
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20, 30],
}

# Initialize GridSearchCV
grid_search = GridSearchCV(
    estimator=rf_classifier,
    param_grid=param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

# Fit the grid search to the data
grid_search.fit(X_train_BOW, y_train)

# Get the best estimator from grid search
best_rf_classifier = grid_search.best_estimator_

# Make predictions using the best model
y_pred = best_rf_classifier.predict(X_test_BOW)

# Evaluate the model
evaluation_results = evaluate_model(
    'RandomForestClassifier - BOW',
    y_test,
    y_pred
)

# Print the evaluation results
for key, value in evaluation_results.items():

    if key == 'Classification Report':
        print(value)

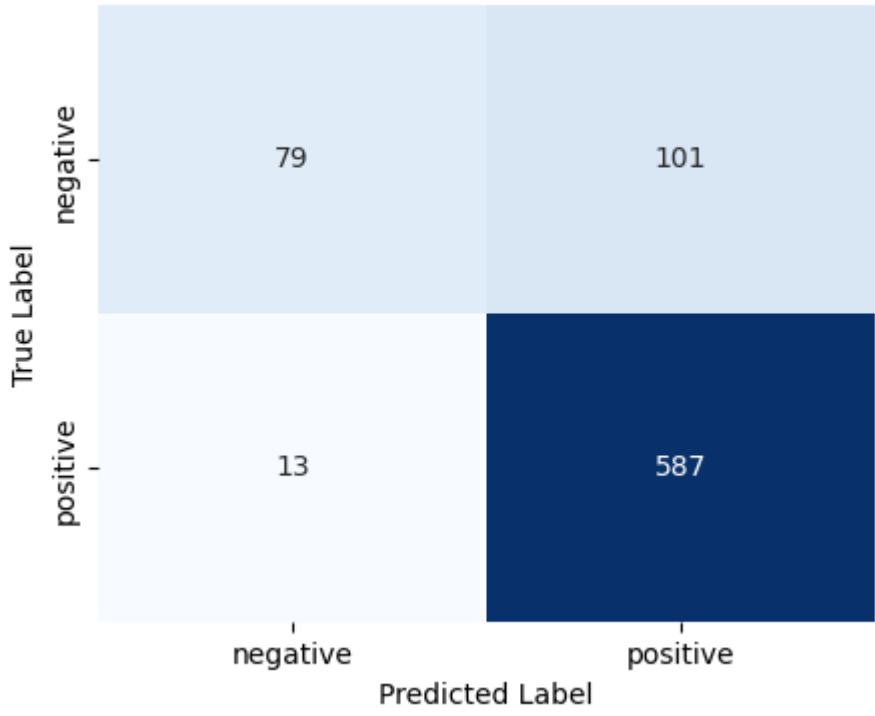
    else:
        print(
            f"{key}: {value:.4f}"
            if isinstance(value, float)
            else f"{key}: \n{value}"
        )

# Print the best parameters found by grid search
print("\nBest hyperparameters found by GridSearchCV:")

print(grid_search.best_params_)

```

Confusion Matrix for RandomForestClassifier - BOW



Model Name:
RandomForestClassifier - BOW
Accuracy: 0.8538
Precision: 0.8545
Recall: 0.8538
F1 Score: 0.8352

	precision	recall	f1-score	support
negative	0.86	0.44	0.58	180
positive	0.85	0.98	0.91	600
accuracy			0.85	780
macro avg	0.86	0.71	0.75	780
weighted avg	0.85	0.85	0.84	780

Best hyperparameters found by GridSearchCV:
{'max_depth': None, 'n_estimators': 300}

```
In [ ]: # CELL 17
# Note: Define, train, and evaluate the Logistic Regression model
# using the Bag of Words feature representation.

# Define the Logistic Regression model
LR = LogisticRegression(random_state=42)

# Define the hyperparameters for grid search
param_grid = {
    'solver': ['liblinear', 'lbfgs'] # Solver type
}

# Initialize GridSearchCV
grid_search = GridSearchCV(
    estimator=LR,
    param_grid=param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

# Fit the grid search to the data
grid_search.fit(X_train_BOW, y_train)

# Get the best estimator from grid search
best_LR = grid_search.best_estimator_

# Make predictions using the best model
y_pred = best_LR.predict(X_test_BOW)

# Evaluate the model
evaluation_results = evaluate_model(
    'LogisticRegression - BOW',
    y_test,
    y_pred
)

# Print the evaluation results
for key, value in evaluation_results.items():

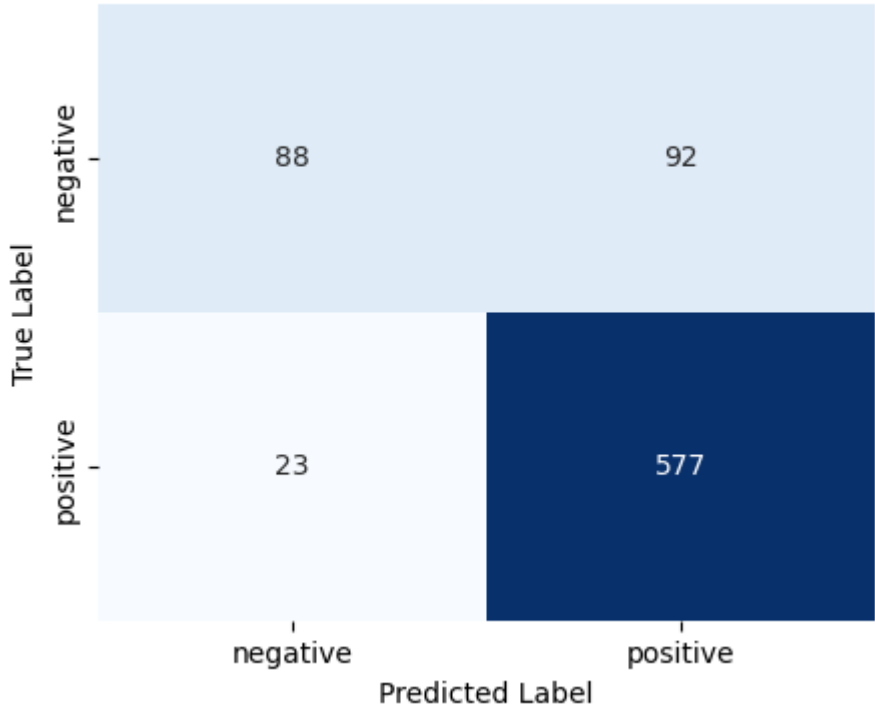
    if key == 'Classification Report':
        print(value)

    else:
        print(
            f"{key}: {value:.4f}"
            if isinstance(value, float)
            else f"{key}: \n{value}"
        )

# Print the best hyperparameters found by grid search
print("\nBest hyperparameters found by GridSearchCV:")

print(grid_search.best_params_)
```

Confusion Matrix for LogisticRegression - BOW



Model Name:
LogisticRegression - BOW
Accuracy: 0.8526
Precision: 0.8464
Recall: 0.8526
F1 Score: 0.8391

	precision	recall	f1-score	support
negative	0.79	0.49	0.60	180
positive	0.86	0.96	0.91	600
accuracy			0.85	780
macro avg	0.83	0.73	0.76	780
weighted avg	0.85	0.85	0.84	780

Best hyperparameters found by GridSearchCV:
{'solver': 'liblinear'}

```
In [ ]: # CELL 18
# Note: Define, train, and evaluate the Decision Tree model
# using the Bag of Words feature representation.

# Define the Decision Tree model
DT = DecisionTreeClassifier(random_state=42)

# Define the hyperparameters for grid search
param_grid = {
    'criterion': ['gini', 'entropy'], # The function to measure the quality of a split
    'max_depth': [None, 10, 20, 30], # The maximum depth of the tree
}

# Initialize GridSearchCV
grid_search = GridSearchCV(
    estimator=DT,
    param_grid=param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

# Fit the grid search to the data
grid_search.fit(X_train_BOW, y_train)

# Get the best estimator from grid search
best_DT = grid_search.best_estimator_

# Make predictions using the best model
y_pred = best_DT.predict(X_test_BOW)

# Evaluate the model
evaluation_results = evaluate_model(
    'DecisionTreeClassifier - BOW',
    y_test,
    y_pred
)

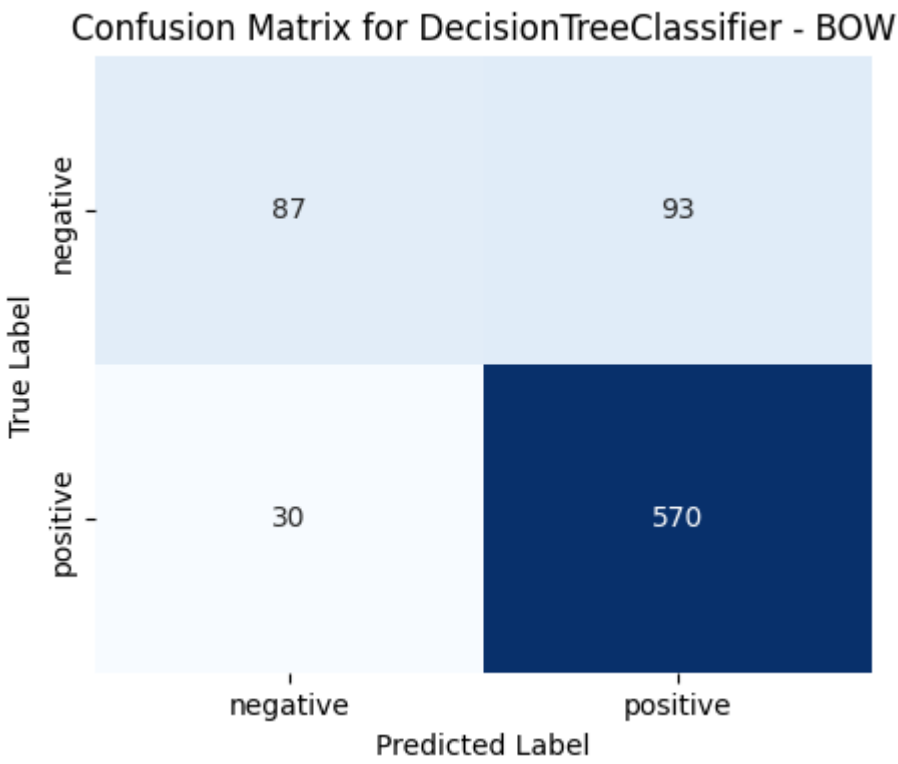
# Print the evaluation results
for key, value in evaluation_results.items():

    if key == 'Classification Report':
        print(value)

    else:
        print(
            f"{key}: {value:.4f}"
            if isinstance(value, float)
            else f"{key}: \n{value}"
        )

# Print the best hyperparameters found by grid search
print("\nBest hyperparameters found by GridSearchCV:")

print(grid_search.best_params_)
```



Model Name:
DecisionTreeClassifier - BOW
Accuracy: 0.8423
Precision: 0.8329
Recall: 0.8423
F1 Score: 0.8295

	precision	recall	f1-score	support
negative	0.74	0.48	0.59	180
positive	0.86	0.95	0.90	600
accuracy			0.84	780
macro avg	0.80	0.72	0.74	780
weighted avg	0.83	0.84	0.83	780

Best hyperparameters found by GridSearchCV:
{'criterion': 'gini', 'max_depth': 30}

```
In [ ]: # CELL 19
# Note: Define, train, and evaluate the Support Vector Machine model
# using the Bag of Words feature representation.

# Define the Support Vector Machine model
SVM = SVC(random_state=42)

# Define the hyperparameters for grid search
param_grid = {
```

```
'kernel': ['linear', 'rbf', 'poly'] # Kernel type
}

# Initialize GridSearchCV
grid_search = GridSearchCV(
    estimator=SVM,
    param_grid=param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

# Fit the grid search to the data
grid_search.fit(X_train_BOW, y_train)

# Get the best estimator from grid search
best_SVM = grid_search.best_estimator_

# Make predictions using the best model
y_pred = best_SVM.predict(X_test_BOW)

# Evaluate the model
evaluation_results = evaluate_model(
    'Support Vector Machine - BOW',
    y_test,
    y_pred
)

# Print the evaluation results
for key, value in evaluation_results.items():

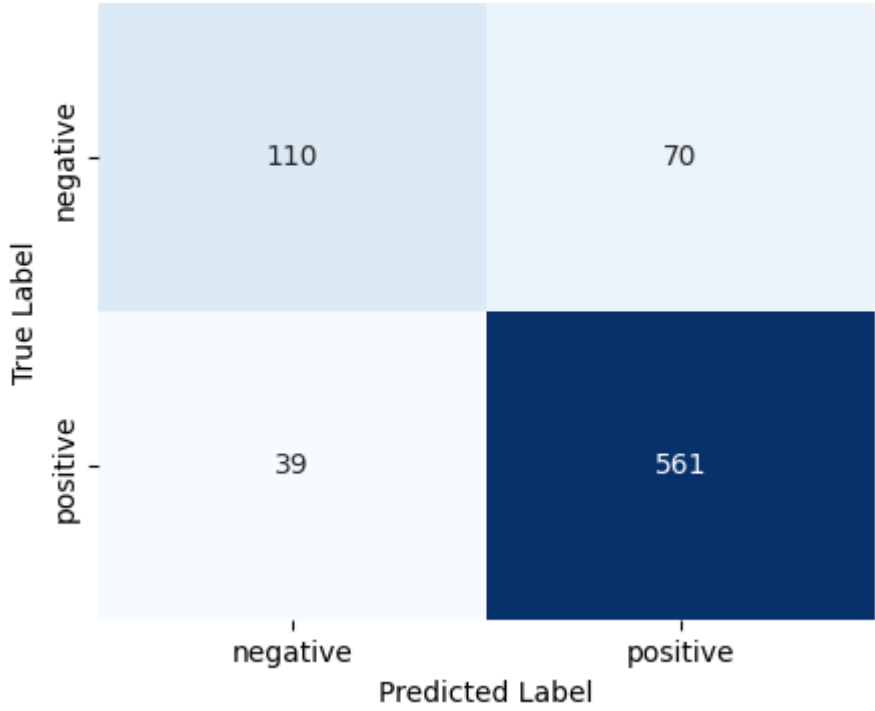
    if key == 'Classification Report':
        print(value)

    else:
        print(
            f"{key}: {value:.4f}"
            if isinstance(value, float)
            else f"{key}: \n{value}"
        )

# Print the best hyperparameters found by grid search
print("\nBest hyperparameters found by GridSearchCV:")

print(grid_search.best_params_)
```

Confusion Matrix for Support Vector Machine - BOW



Model Name:
Support Vector Machine - BOW
Accuracy: 0.8603
Precision: 0.8543
Recall: 0.8603
F1 Score: 0.8554

	precision	recall	f1-score	support
negative	0.74	0.61	0.67	180
positive	0.89	0.94	0.91	600
accuracy			0.86	780
macro avg	0.81	0.77	0.79	780
weighted avg	0.85	0.86	0.86	780

Best hyperparameters found by GridSearchCV:
{'kernel': 'linear'}

```
In [ ]: # CELL 20
# Note: Define, train, and evaluate the K-Nearest Neighbors model
# using the Bag of Words feature representation.

# Define the K-Nearest Neighbors model
knn = KNeighborsClassifier()

# Define the hyperparameters for grid search
param_grid = {
    'n_neighbors': [1, 3, 5, 7, 9],           # Number of neighbors to use
    'weights': ['uniform', 'distance'],       # Weight function used in prediction
    'metric': ['euclidean', 'manhattan'],     # Distance metric
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'] # Algorithm to compute the nearest neighbors
}

# Initialize GridSearchCV
grid_search = GridSearchCV(
    estimator=knn,
    param_grid=param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

# Fit the grid search to the data
grid_search.fit(X_train_BOW, y_train)

# Get the best estimator from grid search
best_knn = grid_search.best_estimator_

# Make predictions using the best model
y_pred = best_knn.predict(X_test_BOW)

# Evaluate the model
evaluation_results = evaluate_model(
    'K-Nearest Neighbors - BOW',
    y_test,
    y_pred
)
```

```
)

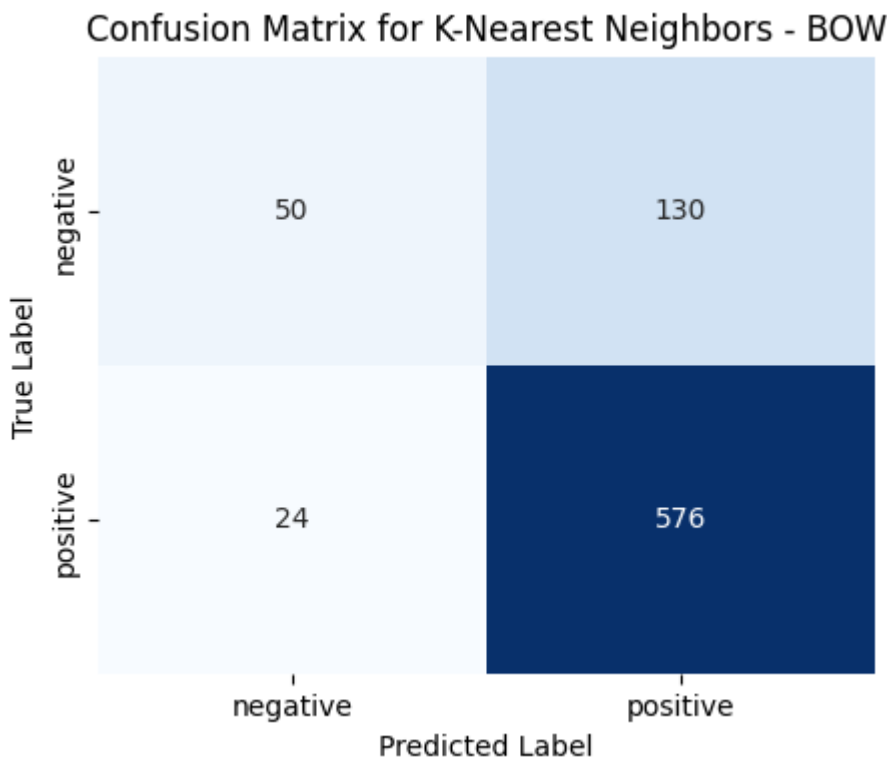
# Print the evaluation results
for key, value in evaluation_results.items():

    if key == 'Classification Report':
        print(value)

    else:
        print(
            f"{key}: {value:.4f}"
            if isinstance(value, float)
            else f"{key}: \n{value}"
        )

# Print the best hyperparameters found by GridSearchCV
print("\nBest hyperparameters found by GridSearchCV:")

print(grid_search.best_params_)
```



Model Name:
K-Nearest Neighbors - BOW
Accuracy: 0.8026
Precision: 0.7835
Recall: 0.8026
F1 Score: 0.7694

	precision	recall	f1-score	support
negative	0.68	0.28	0.39	180
positive	0.82	0.96	0.88	600
accuracy	0.80			780
macro avg	0.75	0.62	0.64	780
weighted avg	0.78	0.80	0.77	780

Best hyperparameters found by GridSearchCV:
{'algorithm': 'auto', 'metric': 'manhattan', 'n_neighbors': 3, 'weights': 'uniform'}

```
In [ ]: # CELL 21
# Note: Apply TF-IDF feature representation using Unigrams.
# Unigrams represent individual words in the text.

vectorizer = TfidfVectorizer(
    ngram_range=(1, 1)
)

X_train_Unigram = vectorizer.fit_transform(X_train)

X_test_Unigram = vectorizer.transform(X_test)
```

```
In [ ]: # CELL 22
# Note: Apply TF-IDF feature representation using Unigrams and Bigrams.
# ngram_range=(1, 2) captures individual words and two-word sequences,
# matching the representation used in the Lecture.

vectorizer2 = TfidfVectorizer(
    ngram_range=(1, 2)
)

X_train_TFIDF = vectorizer2.fit_transform(X_train)

X_test_TFIDF = vectorizer2.transform(X_test)
```

```
In [ ]: # CELL 23
# Note: Apply TF-IDF feature representation using Unigrams, Bigrams, and Trigrams.
# This representation captures individual words and sequences of up to three words.

vectorizer3 = TfidfVectorizer(
    ngram_range=(1, 3)
)

X_train_U_B_T_nigram = vectorizer3.fit_transform(X_train)

X_test_U_B_T_nigram = vectorizer3.transform(X_test)
```

```
In [ ]: # CELL 24
# Note: Check the dimensions of the TF-IDF matrices
# for Unigrams, Unigrams + Bigrams, and Unigrams + Bigrams + Trigrams.

print("Unigram:")
print(X_train_Unigram.shape)
print(X_test_Unigram.shape)

print("\nUnigrams + Bigrams:")
print(X_train_TFIDF.shape)
print(X_test_TFIDF.shape)

print("\nUnigrams + Bigrams + Trigrams:")
print(X_train_U_B_T_nigram.shape)
print(X_test_U_B_T_nigram.shape)
```

Unigram:
(1820, 3657)
(780, 3657)

Unigrams + Bigrams:
(1820, 13909)
(780, 13909)

Unigrams + Bigrams + Trigrams:
(1820, 23462)
(780, 23462)

```
In [ ]: # CELL 25
# Note: Train and evaluate Logistic Regression using TF-IDF
# with Unigrams and Bigrams.

# Define the Logistic Regression model
LR = LogisticRegression(random_state=42)

# Define the hyperparameters for grid search
param_grid = {
    'solver': ['liblinear', 'lbfgs']
}

# Initialize GridSearchCV
grid_search = GridSearchCV(
    estimator=LR,
    param_grid=param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

# Fit the grid search to the data
grid_search.fit(X_train_TFIDF, y_train)

# Get the best estimator from grid search
best_LR_TFIDF = grid_search.best_estimator_

# Make predictions using the best model
y_pred = best_LR_TFIDF.predict(X_test_TFIDF)

# Evaluate the model
evaluation_results = evaluate_model(
    'LogisticRegression - TF-IDF',
    y_test,
    y_pred
)

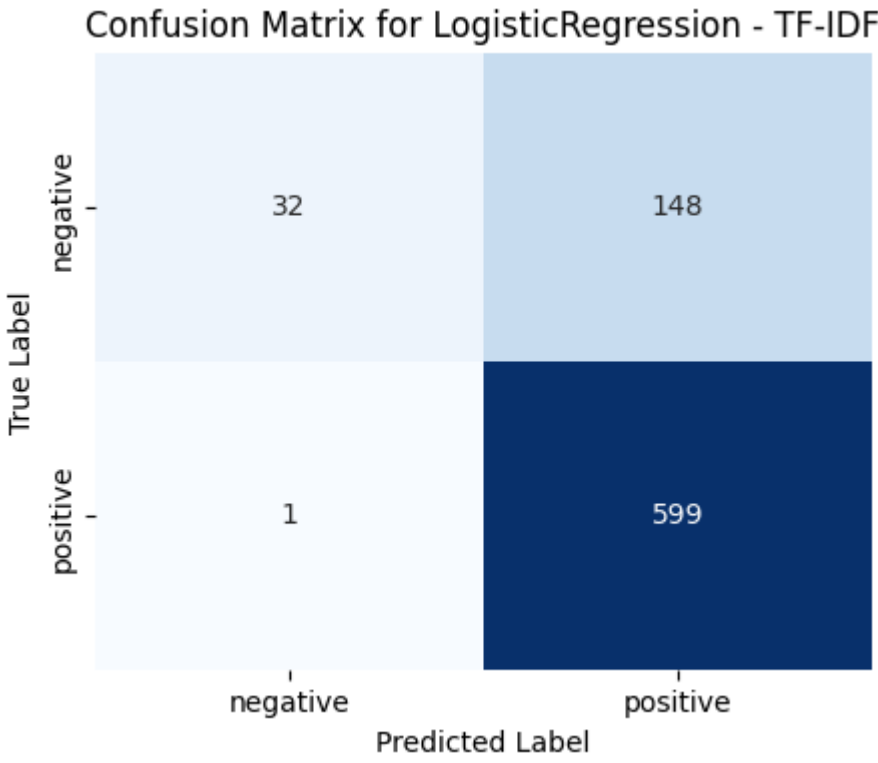
# Print the evaluation results
for key, value in evaluation_results.items():

    if key == 'Classification Report':
        print(value)

    else:
        print(
            f"{key}: {value:.4f}"
            if isinstance(value, float)
            else f"{key}: \n{value}"
        )

# Print the best hyperparameters found by GridSearchCV
print("\nBest hyperparameters found by GridSearchCV:")

print(grid_search.best_params_)
```



Model Name:
LogisticRegression - TF-IDF
Accuracy: 0.8090
Precision: 0.8406
Recall: 0.8090
F1 Score: 0.7535

	precision	recall	f1-score	support
negative	0.97	0.18	0.30	180
positive	0.80	1.00	0.89	600
accuracy			0.81	780
macro avg	0.89	0.59	0.59	780
weighted avg	0.84	0.81	0.75	780

Best hyperparameters found by GridSearchCV:
{'solver': 'liblinear'}

```
In [ ]: # CELL 26
# Note: Train and evaluate Random Forest using TF-IDF
# with Unigrams and Bigrams.

# Define the Random Forest Classifier
rf_classifier = RandomForestClassifier(random_state=42)

# Define the hyperparameters for grid search
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20, 30],
}

# Initialize GridSearchCV
grid_search = GridSearchCV(
    estimator=rf_classifier,
    param_grid=param_grid,
```

```
cv=5,
scoring='accuracy',
n_jobs=-1
)

# Fit the grid search to the data
grid_search.fit(X_train_TFIDF, y_train)

# Get the best estimator from grid search
best_rf_TFIDF = grid_search.best_estimator_

# Make predictions using the best model
y_pred = best_rf_TFIDF.predict(X_test_TFIDF)

# Evaluate the model
evaluation_results = evaluate_model(
    'RandomForestClassifier - TF-IDF',
    y_test,
    y_pred
)

# Print the evaluation results
for key, value in evaluation_results.items():

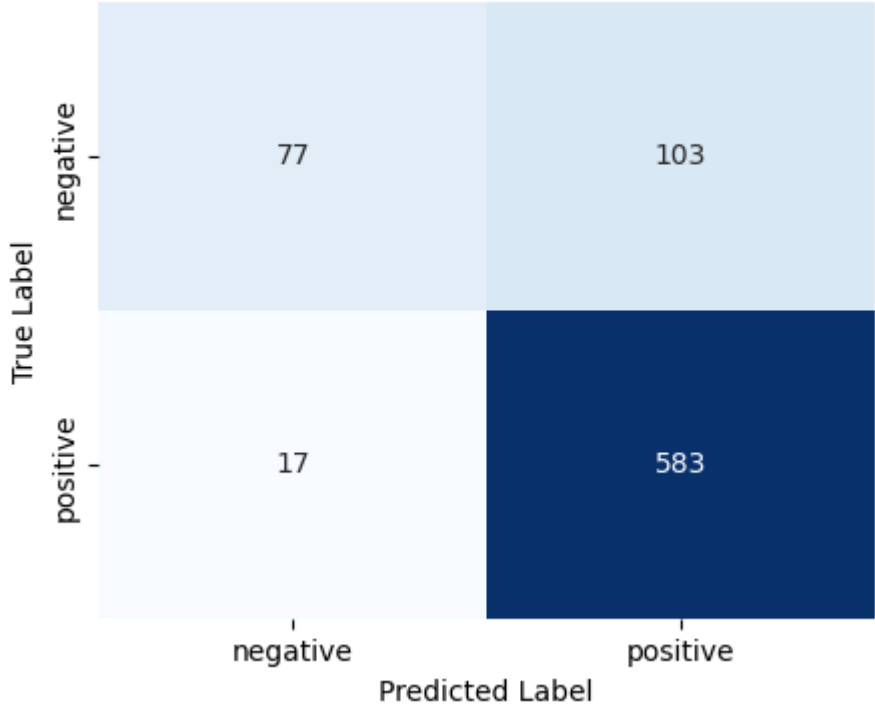
    if key == 'Classification Report':
        print(value)

    else:
        print(
            f"{key}: {value:.4f}"
            if isinstance(value, float)
            else f"{key}: \n{value}"
        )

# Print the best hyperparameters found by GridSearchCV
print("\nBest hyperparameters found by GridSearchCV:")

print(grid_search.best_params_)
```

Confusion Matrix for RandomForestClassifier - TF-IDF



Model Name:
RandomForestClassifier - TF-IDF
Accuracy: 0.8462
Precision: 0.8428
Recall: 0.8462
F1 Score: 0.8272

	precision	recall	f1-score	support
negative	0.82	0.43	0.56	180
positive	0.85	0.97	0.91	600
accuracy			0.85	780
macro avg	0.83	0.70	0.73	780
weighted avg	0.84	0.85	0.83	780

Best hyperparameters found by GridSearchCV:
{'max_depth': None, 'n_estimators': 300}

```
In [ ]: # CELL 27
# Note: Train and evaluate Decision Tree using TF-IDF
# with Unigrams and Bigrams.

# Define the Decision Tree model
DT = DecisionTreeClassifier(random_state=42)

# Define the hyperparameters for grid search
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 10, 20, 30],
}

# Initialize GridSearchCV
grid_search = GridSearchCV(
    estimator=DT,
    param_grid=param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

# Fit the grid search to the data
grid_search.fit(X_train_TFIDF, y_train)

# Get the best estimator from grid search
best_DT_TFIDF = grid_search.best_estimator_

# Make predictions using the best model
y_pred = best_DT_TFIDF.predict(X_test_TFIDF)

# Evaluate the model
evaluation_results = evaluate_model(
    'DecisionTreeClassifier - TF-IDF',
    y_test,
    y_pred
)

# Print the evaluation results
for key, value in evaluation_results.items():

    if key == 'Classification Report':
        print(value)

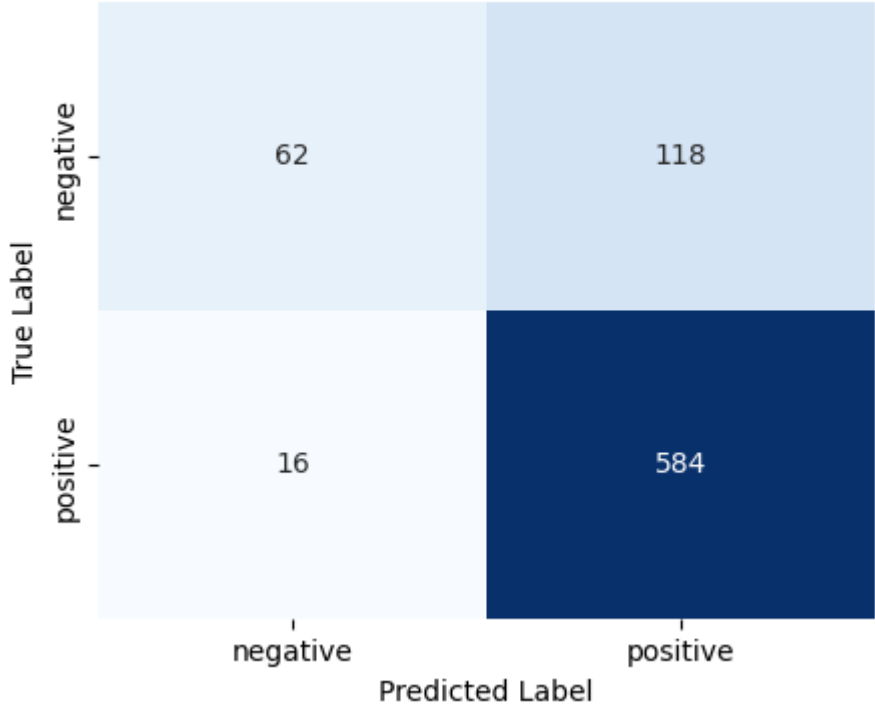
    else:
```

```
print(
    f"{key}: {value:.4f}"
    if isinstance(value, float)
    else f"{key}: \n{value}"
)

# Print the best hyperparameters found by GridSearchCV
print("\nBest hyperparameters found by GridSearchCV:")

print(grid_search.best_params_)
```

Confusion Matrix for DecisionTreeClassifier - TF-IDF



Model Name:
DecisionTreeClassifier - TF-IDF
Accuracy: 0.8282
Precision: 0.8234
Recall: 0.8282
F1 Score: 0.8010

	precision	recall	f1-score	support
negative	0.79	0.34	0.48	180
positive	0.83	0.97	0.90	600
accuracy			0.83	780
macro avg	0.81	0.66	0.69	780
weighted avg	0.82	0.83	0.80	780

Best hyperparameters found by GridSearchCV:
{'criterion': 'entropy', 'max_depth': 20}

```
In [ ]: # CELL 28
# Note: Train and evaluate Support Vector Machine using TF-IDF
# with Unigrams and Bigrams.

# Define the Support Vector Machine model
SVM = SVC(random_state=42)

# Define the hyperparameters for grid search
param_grid = {
    'kernel': ['linear', 'rbf', 'poly']
}

# Initialize GridSearchCV
grid_search = GridSearchCV(
    estimator=SVM,
    param_grid=param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

# Fit the grid search to the data
grid_search.fit(X_train_TFIDF, y_train)

# Get the best estimator from grid search
best_SVM_TFIDF = grid_search.best_estimator_

# Make predictions using the best model
y_pred = best_SVM_TFIDF.predict(X_test_TFIDF)

# Evaluate the model
evaluation_results = evaluate_model(
    'Support Vector Machine - TF-IDF',
    y_test,
    y_pred
)

# Print the evaluation results
for key, value in evaluation_results.items():

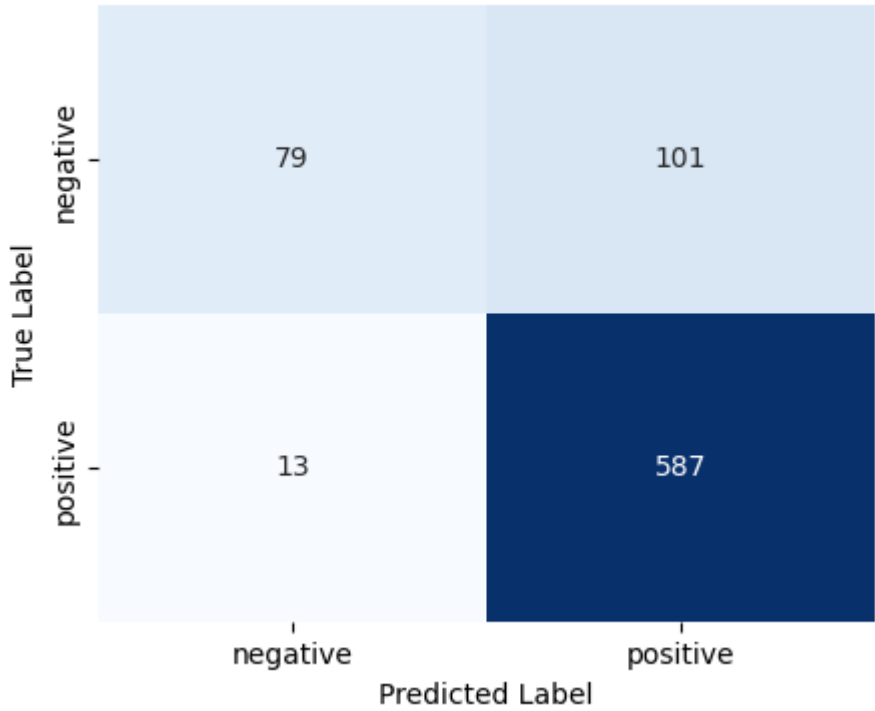
    if key == 'Classification Report':
        print(value)

    else:
        print(
            f"{key}: {value:.4f}"
            if isinstance(value, float)
            else f"{key}: \n{value}"
        )

# Print the best hyperparameters found by GridSearchCV
print("\nBest hyperparameters found by GridSearchCV:")

print(grid_search.best_params_)
```

Confusion Matrix for Support Vector Machine - TF-IDF



Model Name:
Support Vector Machine - TF-IDF
Accuracy: 0.8538
Precision: 0.8545
Recall: 0.8538
F1 Score: 0.8352

	precision	recall	f1-score	support
negative	0.86	0.44	0.58	180
positive	0.85	0.98	0.91	600
accuracy			0.85	780
macro avg	0.86	0.71	0.75	780
weighted avg	0.85	0.85	0.84	780

Best hyperparameters found by GridSearchCV:
{'kernel': 'linear'}

```
In [ ]: # CELL 29
# Note: Train and evaluate K-Nearest Neighbors using TF-IDF
# with Unigrams, Bigrams, and Trigrams.

# Define the K-Nearest Neighbors model
knn = KNeighborsClassifier()

# Define the hyperparameters for grid search
param_grid = {
    'n_neighbors': [1, 3, 5, 7, 9],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan'],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute']
}

# Initialize GridSearchCV
grid_search = GridSearchCV(
    estimator=knn,
    param_grid=param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

# Fit the grid search to the data
grid_search.fit(X_train_U_B_T_nigram, y_train)

# Get the best estimator from grid search
best_knn_TFIDF = grid_search.best_estimator_

# Make predictions using the best model
y_pred = best_knn_TFIDF.predict(X_test_U_B_T_nigram)

# Evaluate the model
evaluation_results = evaluate_model(
    'K-Nearest Neighbors - TF-IDF',
    y_test,
    y_pred
)

# Print the evaluation results
for key, value in evaluation_results.items():

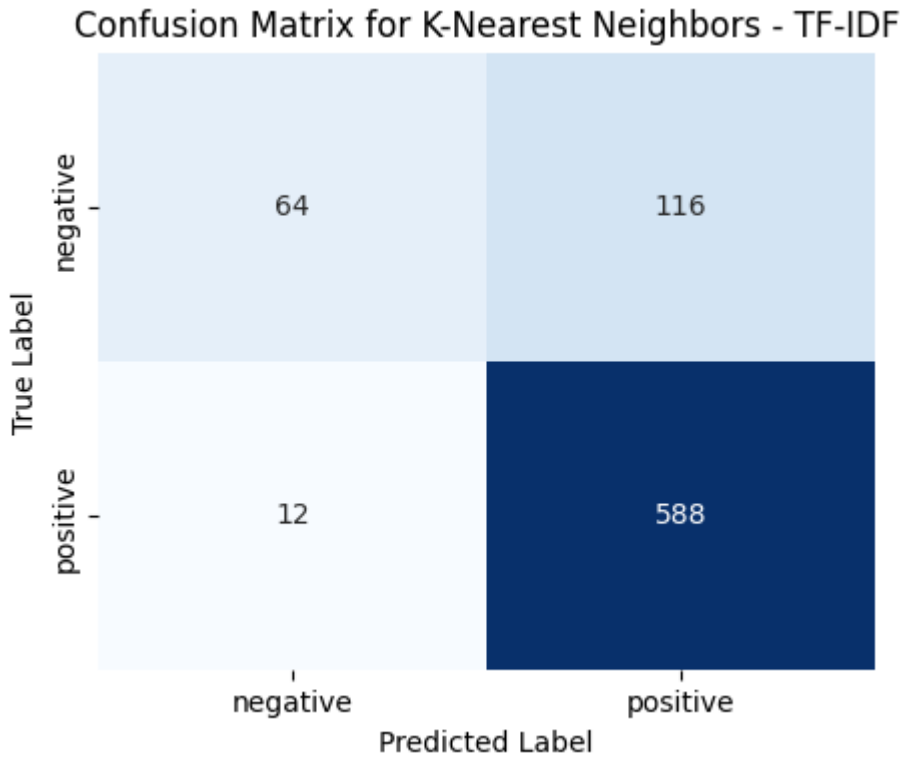
    if key == 'Classification Report':
        print(value)

    else:
        print(
            f"{key}: {value:.4f}"
            if isinstance(value, float)
            else f"{key}: \n{value}"
        )

# Print the best hyperparameters found by GridSearchCV
print("\nBest hyperparameters found by GridSearchCV:")

print(grid_search.best_params_)
```

/usr/local/lib/python3.13/dist-packages/sklearn/model_selection/_search.py:1108: UserWarning: One or more of the test scores are non-finite: [0.77252747 0.77252747 0.81593407 0.81593407 0.82252747 0.82307692 0.82362637 0.82417582 0.81758242 0.81813187 nan 0.77032967 nan 0.77472527 nan 0.76483516 nan 0.76648352 nan 0.76923077 0.77252747 0.77252747 0.81593407 0.81593407 0.82252747 0.82307692 0.82362637 0.82417582 0.81758242 0.81813187 nan 0.77032967 nan 0.76483516 nan 0.76648352 0.81593407 0.82252747 0.82307692 0.82362637 0.82417582 0.81758242 0.81813187 nan 0.77032967 nan 0.77472527 nan 0.76483516 nan 0.76648352 0.76923077]
warnings.warn(



Model Name:
K-Nearest Neighbors - TF-IDF
Accuracy: 0.8359
Precision: 0.8368
Recall: 0.8359
F1 Score: 0.8091

	precision	recall	f1-score	support
negative	0.84	0.36	0.50	180
positive	0.84	0.98	0.90	600
accuracy			0.84	780
macro avg	0.84	0.67	0.70	780
weighted avg	0.84	0.84	0.81	780

Best hyperparameters found by GridSearchCV:
{'algorithm': 'auto', 'metric': 'euclidean', 'n_neighbors': 7, 'weights': 'distance'}

```
In [ ]: # CELL 30
# Note: What is an imbalanced dataset?
# An imbalanced dataset is one in which the classes are not represented
# equally - one class (here: positive) has significantly more samples than
# the other (negative). This can bias the model toward the majority class.
#
# Resampling techniques used below to address this:
# 1. RandomOverSampler - duplicates minority class samples.
# 2. RandomUnderSampler - removes majority class samples.
# 3. SMOTE - generates synthetic minority class samples.
#
# Uncomment the line below the first time you run this in Colab:
# !pip install imbalanced-learn

from imblearn.over_sampling import RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import SMOTE
from collections import Counter
```

```
In [ ]: # CELL 31
# Note: Define a reusable function to train a Random Forest with GridSearchCV
# on any given (X, y) training data and evaluate it on the fixed test set.
# This mirrors the helper function used in the Lecture so the same model
# can easily be retrained on the baseline data and on each resampled version.

def train_random_forest_classifier_with_grid_search(
    X_train_vec,
    y_train_vec,
    X_test_vec,
    y_test_vec,
    evaluate_model_func
):
    # Define the Random Forest Classifier
    rf_classifier = RandomForestClassifier(random_state=42)

    # Define the hyperparameters for grid search
    param_grid = {
        'n_estimators': [100, 200, 300],
        'max_depth': [None, 10, 20, 30],
    }

    # Initialize GridSearchCV
    grid_search = GridSearchCV(
        estimator=rf_classifier,
        param_grid=param_grid,
        cv=5,
        scoring='accuracy',
        n_jobs=-1
    )

    # Fit the grid search to the data
    grid_search.fit(X_train_vec, y_train_vec)

    # Get the best estimator from grid search
    best_rf = grid_search.best_estimator_

    # Make predictions using the best model
    y_pred = best_rf.predict(X_test_vec)

    # Evaluate the model
    evaluation_results = evaluate_model_func('RandomForestClassifier', y_test_vec, y_pred)

    # Print the evaluation results
    for key, value in evaluation_results.items():
        if key == 'Classification Report':
            print(value)
        else:
            print(
                f"{key}: {value:.4f}"
                if isinstance(value, float)
                else f"{key}: \n{value}"
            )

    # Print the best parameters found by grid search
    print("\nBest hyperparameters found by GridSearchCV:")
    print(grid_search.best_params_)

    return best_rf, evaluation_results
```

```
In [ ]: # CELL 32
# Note: Apply RandomOverSampler on the Bag of Words training data
# to balance the classes by duplicating minority (negative) samples.

ros = RandomOverSampler(random_state=42)

X_ros, y_ros = ros.fit_resample(X_train_BOW, y_train)

print("After RandomOversampling:", Counter(y_ros))
```

After RandomOversampling: Counter({'positive': 1400, 'negative': 1400})

```
In [ ]: # CELL 33
# Note: Apply RandomUnderSampler on the Bag of Words training data
# to balance the classes by removing majority (positive) samples.

rus = RandomUnderSampler(random_state=42)

X_rus, y_rus = rus.fit_resample(X_train_BOW, y_train)

print("After RandomUndersampling:", Counter(y_rus))
```

After RandomUndersampling: Counter({'negative': 420, 'positive': 420})

```
In [ ]: # CELL 34
# Note: Apply SMOTE on the Bag of Words training data to balance the
# classes by generating synthetic minority (negative) samples.

smote = SMOTE(random_state=42)

X_smote, y_smote = smote.fit_resample(X_train_BOW, y_train)

print("After SMOTE:", Counter(y_smote))
```

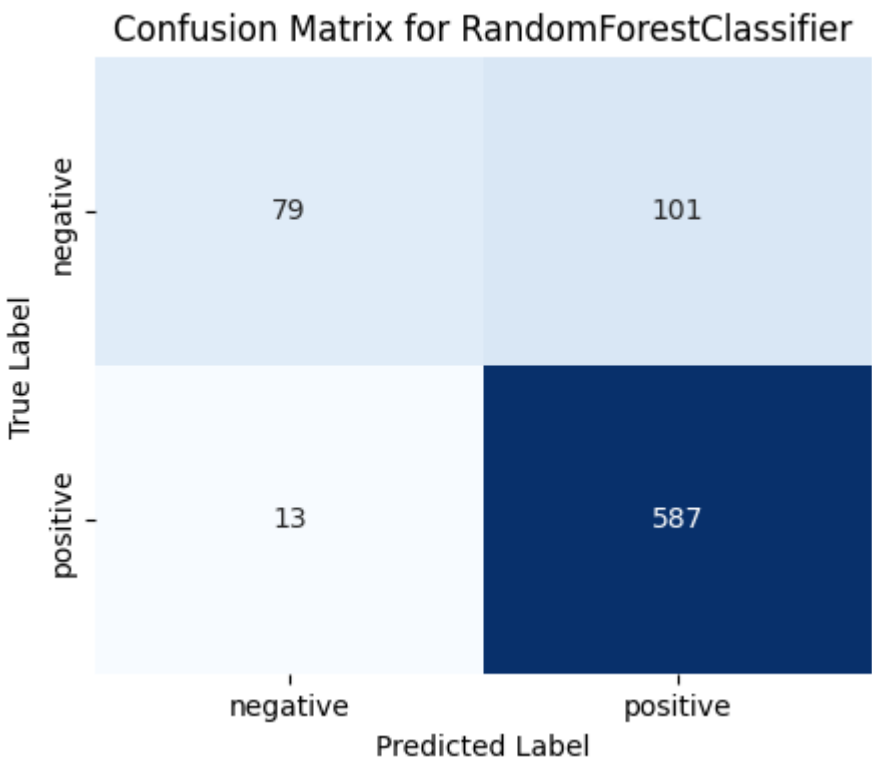
After SMOTE: Counter({'positive': 1400, 'negative': 1400})

```
In [ ]: # CELL 35
# Note: Train a Random Forest on the original (baseline) imbalanced
# training data as a reference point for comparison.

print("Random Forest - Baseline (No Resampling)")
print("=====")

best_rf_baseline, results_baseline = train_random_forest_classifier_with_grid_search(
    X_train_BOW, y_train, X_test_BOW, y_test, evaluate_model
)
```

Random Forest - Baseline (No Resampling)
=====



Model Name:
RandomForestClassifier
Accuracy: 0.8538
Precision: 0.8545
Recall: 0.8538
F1 Score: 0.8352

	precision	recall	f1-score	support
negative	0.86	0.44	0.58	180
positive	0.85	0.98	0.91	600
accuracy			0.85	780
macro avg	0.86	0.71	0.75	780
weighted avg	0.85	0.85	0.84	780

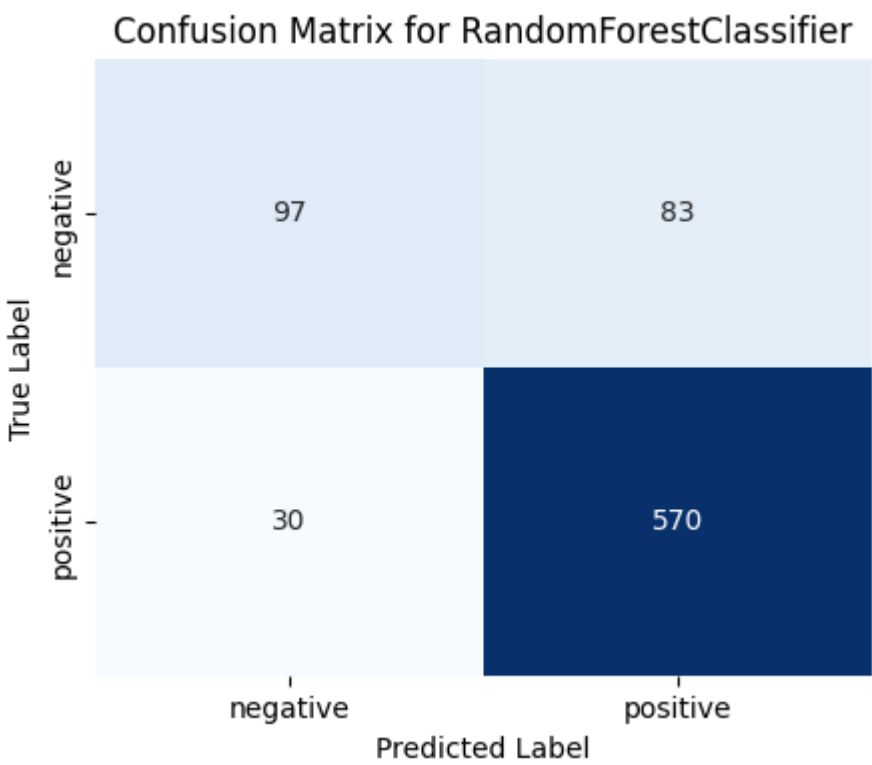
Best hyperparameters found by GridSearchCV:
{'max_depth': None, 'n_estimators': 300}

```
In [ ]: # CELL 36
# Note: Train a Random Forest on the RandomOverSampler-resampled data.

print("Random Forest - RandomOverSampler")
print("=====")

best_rf_ros, results_ros = train_random_forest_classifier_with_grid_search(
    X_ros, y_ros, X_test_BOW, y_test, evaluate_model
)
```

Random Forest - RandomOverSampler
=====



Model Name:
RandomForestClassifier
Accuracy: 0.8551
Precision: 0.8477
Recall: 0.8551
F1 Score: 0.8457

	precision	recall	f1-score	support
negative	0.76	0.54	0.63	180
positive	0.87	0.95	0.91	600
accuracy			0.86	780
macro avg	0.82	0.74	0.77	780
weighted avg	0.85	0.86	0.85	780

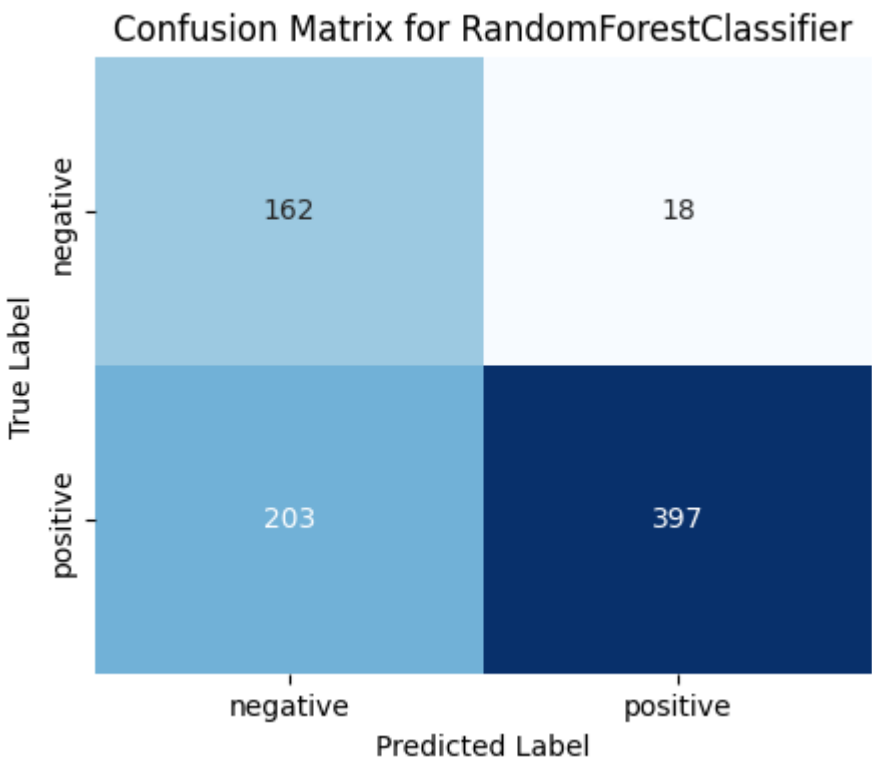
Best hyperparameters found by GridSearchCV:
{'max_depth': None, 'n_estimators': 200}

```
In [ ]: # CELL 37
# Note: Train a Random Forest on the RandomUnderSampler-resampled data.

print("Random Forest - RandomUnderSampler")
print("=====")

best_rf_rus, results_rus = train_random_forest_classifier_with_grid_search(
    X_rus, y_rus, X_test_BOW, y_test, evaluate_model
)
```

Random Forest - RandomUnderSampler
=====



Model Name:
RandomForestClassifier
Accuracy: 0.7167
Precision: 0.8383
Recall: 0.7167
F1 Score: 0.7389

	precision	recall	f1-score	support
negative	0.44	0.90	0.59	180
positive	0.96	0.66	0.78	600
accuracy			0.72	780
macro avg	0.70	0.78	0.69	780
weighted avg	0.84	0.72	0.74	780

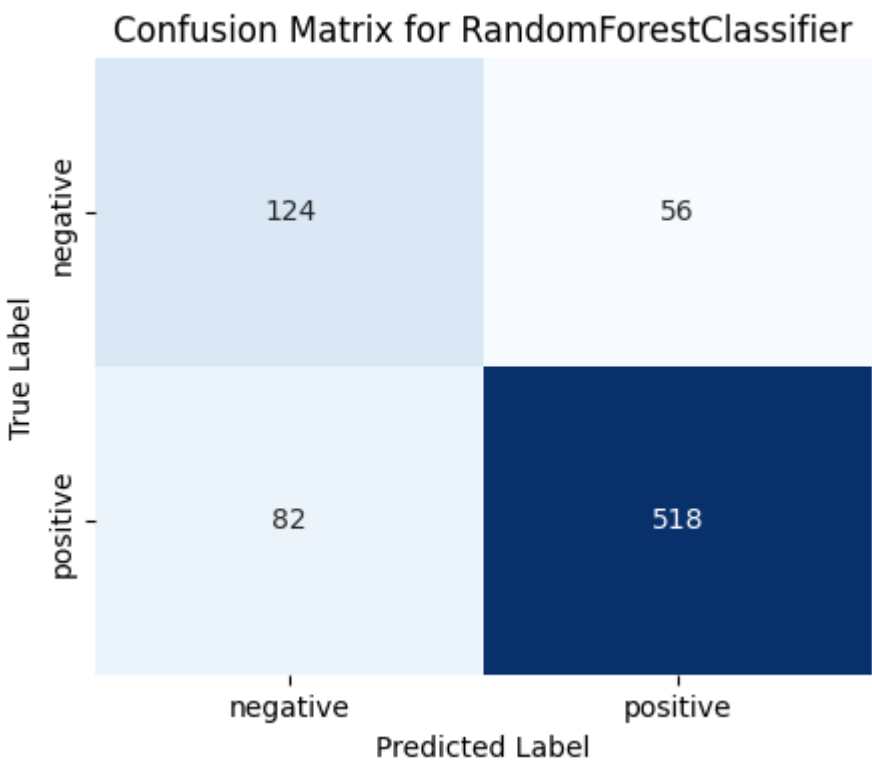
Best hyperparameters found by GridSearchCV:
{'max_depth': 10, 'n_estimators': 100}

```
In [ ]: # CELL 38
# Note: Train a Random Forest on the SMOTE-resampled data.

print("Random Forest - SMOTE")
print("=====")

best_rf_smote, results_smote = train_random_forest_classifier_with_grid_search(
    X_smote, y_smote, X_test_BOW, y_test, evaluate_model
)
```

Random Forest - SMOTE
=====



Model Name:
RandomForestClassifier
Accuracy: 0.8231
Precision: 0.8331
Recall: 0.8231
F1 Score: 0.8271

	precision	recall	f1-score	support
negative	0.60	0.69	0.64	180
positive	0.90	0.86	0.88	600
accuracy			0.82	780
macro avg	0.75	0.78	0.76	780
weighted avg	0.83	0.82	0.83	780

Best hyperparameters found by GridSearchCV:
{'max_depth': None, 'n_estimators': 200}

```
In [ ]: # CELL 39
# Note: Compare the baseline model against each resampling technique
# to see the effect of handling the class imbalance.

resampling_results = pd.DataFrame([
    {
        'Technique': 'Baseline (No Resampling)',
        'Accuracy': results_baseline['Accuracy'],
        'Precision': results_baseline['Precision'],
        'Recall': results_baseline['Recall'],
        'F1 Score': results_baseline['F1 Score']
    },
    {
        'Technique': 'RandomOverSampler',
        'Accuracy': results_ros['Accuracy'],
        'Precision': results_ros['Precision'],
        'Recall': results_ros['Recall'],
        'F1 Score': results_ros['F1 Score']
    },
    {
        'Technique': 'RandomUnderSampler',
        'Accuracy': results_rus['Accuracy'],
        'Precision': results_rus['Precision'],
        'Recall': results_rus['Recall'],
        'F1 Score': results_rus['F1 Score']
    },
    {
        'Technique': 'SMOTE',
        'Accuracy': results_smote['Accuracy'],
        'Precision': results_smote['Precision'],
        'Recall': results_smote['Recall'],
        'F1 Score': results_smote['F1 Score']
    }
])

resampling_results
```

Out[]:

	Technique	Accuracy	Precision	Recall	F1 Score
0	Baseline (No Resampling)	0.853846	0.854466	0.853846	0.835196
1	RandomOverSampler	0.855128	0.847714	0.855128	0.845687
2	RandomUnderSampler	0.716667	0.838290	0.716667	0.738934
3	SMOTE	0.823077	0.833093	0.823077	0.827076

```
In [ ]: # CELL 40
# Note: Compare the performance of the classification models
# using the Bag of Words feature representation.

bow_results = []

# Random Forest
y_pred = best_rf_classifier.predict(X_test_BOW)
bow_results.append({
    'Model': 'Random Forest - BOW',
    'Accuracy': accuracy_score(y_test, y_pred),
    'Precision': precision_score(y_test, y_pred, average='weighted'),
    'Recall': recall_score(y_test, y_pred, average='weighted'),
    'F1 Score': f1_score(y_test, y_pred, average='weighted')
})

# Logistic Regression
y_pred = best_LR.predict(X_test_BOW)
bow_results.append({
    'Model': 'Logistic Regression - BOW',
    'Accuracy': accuracy_score(y_test, y_pred),
    'Precision': precision_score(y_test, y_pred, average='weighted'),
    'Recall': recall_score(y_test, y_pred, average='weighted'),
    'F1 Score': f1_score(y_test, y_pred, average='weighted')
})

# Decision Tree
y_pred = best_DT.predict(X_test_BOW)
bow_results.append({
    'Model': 'Decision Tree - BOW',
    'Accuracy': accuracy_score(y_test, y_pred),
    'Precision': precision_score(y_test, y_pred, average='weighted'),
    'Recall': recall_score(y_test, y_pred, average='weighted'),
    'F1 Score': f1_score(y_test, y_pred, average='weighted')
})

# SVM
y_pred = best_SVM.predict(X_test_BOW)
bow_results.append({
    'Model': 'SVM - BOW',
    'Accuracy': accuracy_score(y_test, y_pred),
    'Precision': precision_score(y_test, y_pred, average='weighted'),
    'Recall': recall_score(y_test, y_pred, average='weighted'),
    'F1 Score': f1_score(y_test, y_pred, average='weighted')
})

# KNN
y_pred = best_knn.predict(X_test_BOW)
bow_results.append({
    'Model': 'KNN - BOW',
    'Accuracy': accuracy_score(y_test, y_pred),
    'Precision': precision_score(y_test, y_pred, average='weighted'),
    'Recall': recall_score(y_test, y_pred, average='weighted'),
    'F1 Score': f1_score(y_test, y_pred, average='weighted')
})

bow_results_df = pd.DataFrame(bow_results)

bow_results_df
```

Out[]:

	Model	Accuracy	Precision	Recall	F1 Score
0	Random Forest - BOW	0.853846	0.854466	0.853846	0.835196
1	Logistic Regression - BOW	0.852564	0.846399	0.852564	0.839093
2	Decision Tree - BOW	0.842308	0.832927	0.842308	0.829516
3	SVM - BOW	0.860256	0.854263	0.860256	0.855432
4	KNN - BOW	0.802564	0.783513	0.802564	0.769379

```
In [ ]: # CELL 41
# Note: Compare the performance of the classification models
# using the TF-IDF feature representation.

tfidf_results = []
```

```
# Logistic Regression
y_pred = best_LR_TFIDF.predict(X_test_TFIDF)
tfidf_results.append({
    'Model': 'Logistic Regression - TF-IDF',
    'Accuracy': accuracy_score(y_test, y_pred),
    'Precision': precision_score(y_test, y_pred, average='weighted'),
    'Recall': recall_score(y_test, y_pred, average='weighted'),
    'F1 Score': f1_score(y_test, y_pred, average='weighted')
})

# Random Forest
y_pred = best_rf_TFIDF.predict(X_test_TFIDF)
tfidf_results.append({
    'Model': 'Random Forest - TF-IDF',
    'Accuracy': accuracy_score(y_test, y_pred),
    'Precision': precision_score(y_test, y_pred, average='weighted'),
    'Recall': recall_score(y_test, y_pred, average='weighted'),
    'F1 Score': f1_score(y_test, y_pred, average='weighted')
})

# Decision Tree
y_pred = best_DT_TFIDF.predict(X_test_TFIDF)
tfidf_results.append({
    'Model': 'Decision Tree - TF-IDF',
    'Accuracy': accuracy_score(y_test, y_pred),
    'Precision': precision_score(y_test, y_pred, average='weighted'),
    'Recall': recall_score(y_test, y_pred, average='weighted'),
    'F1 Score': f1_score(y_test, y_pred, average='weighted')
})

# SVM
y_pred = best_SVM_TFIDF.predict(X_test_TFIDF)
tfidf_results.append({
    'Model': 'SVM - TF-IDF',
    'Accuracy': accuracy_score(y_test, y_pred),
    'Precision': precision_score(y_test, y_pred, average='weighted'),
    'Recall': recall_score(y_test, y_pred, average='weighted'),
    'F1 Score': f1_score(y_test, y_pred, average='weighted')
})

# KNN
y_pred = best_knn_TFIDF.predict(X_test_U_B_T_nigram)
tfidf_results.append({
    'Model': 'KNN - TF-IDF (1-3 grams)',
    'Accuracy': accuracy_score(y_test, y_pred),
    'Precision': precision_score(y_test, y_pred, average='weighted'),
    'Recall': recall_score(y_test, y_pred, average='weighted'),
    'F1 Score': f1_score(y_test, y_pred, average='weighted')
})

tfidf_results_df = pd.DataFrame(tfidf_results)

tfidf_results_df
```

Out[]:

	Model	Accuracy	Precision	Recall	F1 Score
0	Logistic Regression - TF-IDF	0.808974	0.840603	0.808974	0.753481
1	Random Forest - TF-IDF	0.846154	0.842768	0.846154	0.827154
2	Decision Tree - TF-IDF	0.828205	0.823362	0.828205	0.800975
3	SVM - TF-IDF	0.853846	0.854466	0.853846	0.835196
4	KNN - TF-IDF (1-3 grams)	0.835897	0.836815	0.835897	0.809108

```
In [ ]: # CELL 42
# Note: Combine the results of all classification models
# to obtain one final comparison table.

final_results = pd.concat(
    [
        bow_results_df,
        tfidf_results_df
    ],
    ignore_index=True
)

final_results
```

Out[]:

	Model	Accuracy	Precision	Recall	F1 Score
0	Random Forest - BOW	0.853846	0.854466	0.853846	0.835196
1	Logistic Regression - BOW	0.852564	0.846399	0.852564	0.839093
2	Decision Tree - BOW	0.842308	0.832927	0.842308	0.829516
3	SVM - BOW	0.860256	0.854263	0.860256	0.855432
4	KNN - BOW	0.802564	0.783513	0.802564	0.769379
5	Logistic Regression - TF-IDF	0.808974	0.840603	0.808974	0.753481
6	Random Forest - TF-IDF	0.846154	0.842768	0.846154	0.827154
7	Decision Tree - TF-IDF	0.828205	0.823362	0.828205	0.800975
8	SVM - TF-IDF	0.853846	0.854466	0.853846	0.835196
9	KNN - TF-IDF (1-3 grams)	0.835897	0.836815	0.835897	0.809108

```
In [ ]: # CELL 43
# Note: Sort all models according to their F1 Score
# to identify the best performing classification model.

final_results = final_results.sort_values(
    by='F1 Score',
    ascending=False
)

final_results
```

Out[]:

	Model	Accuracy	Precision	Recall	F1 Score
3	SVM - BOW	0.860256	0.854263	0.860256	0.855432
1	Logistic Regression - BOW	0.852564	0.846399	0.852564	0.839093
0	Random Forest - BOW	0.853846	0.854466	0.853846	0.835196
8	SVM - TF-IDF	0.853846	0.854466	0.853846	0.835196
2	Decision Tree - BOW	0.842308	0.832927	0.842308	0.829516
6	Random Forest - TF-IDF	0.846154	0.842768	0.846154	0.827154
9	KNN - TF-IDF (1-3 grams)	0.835897	0.836815	0.835897	0.809108
7	Decision Tree - TF-IDF	0.828205	0.823362	0.828205	0.800975
4	KNN - BOW	0.802564	0.783513	0.802564	0.769379
5	Logistic Regression - TF-IDF	0.808974	0.840603	0.808974	0.753481

In []:

```
# CELL 44
# Note: Identify the best performing model based on the highest F1 Score.

best_model = final_results.iloc[0]

print("Best Model:")
print(best_model)
```

Best Model:
Model SVM - BOW
Accuracy 0.860256
Precision 0.854263
Recall 0.860256
F1 Score 0.855432
Name: 3, dtype: object

In []:

```
# CELL 45
# Note: Display the classification report and confusion matrix
# for the best performing model.

best_model_name = best_model['Model']

print("Best Model:", best_model_name)
print("\nClassification Report:\n")

if best_model_name == 'Random Forest - BOW':
    y_pred_best = best_rf_classifier.predict(X_test_BOW)
elif best_model_name == 'Logistic Regression - BOW':
    y_pred_best = best_LR.predict(X_test_BOW)
elif best_model_name == 'Decision Tree - BOW':
    y_pred_best = best_DT.predict(X_test_BOW)
elif best_model_name == 'SVM - BOW':
    y_pred_best = best_SVM.predict(X_test_BOW)
elif best_model_name == 'KNN - BOW':
    y_pred_best = best_knn.predict(X_test_BOW)
elif best_model_name == 'Logistic Regression - TF-IDF':
    y_pred_best = best_LR_TFIDF.predict(X_test_TFIDF)
elif best_model_name == 'Random Forest - TF-IDF':
    y_pred_best = best_rf_TFIDF.predict(X_test_TFIDF)
elif best_model_name == 'Decision Tree - TF-IDF':
    y_pred_best = best_DT_TFIDF.predict(X_test_TFIDF)
elif best_model_name == 'SVM - TF-IDF':
    y_pred_best = best_SVM_TFIDF.predict(X_test_TFIDF)
elif best_model_name == 'KNN - TF-IDF (1-3 grams)':
    y_pred_best = best_knn_TFIDF.predict(X_test_U_B_T_nigram)

print(classification_report(y_test, y_pred_best))
```

Best Model: SVM - BOW

Classification Report:

	precision	recall	f1-score	support
negative	0.74	0.61	0.67	180
positive	0.89	0.94	0.91	600
accuracy			0.86	780
macro avg	0.81	0.77	0.79	780
weighted avg	0.85	0.86	0.86	780

In []:

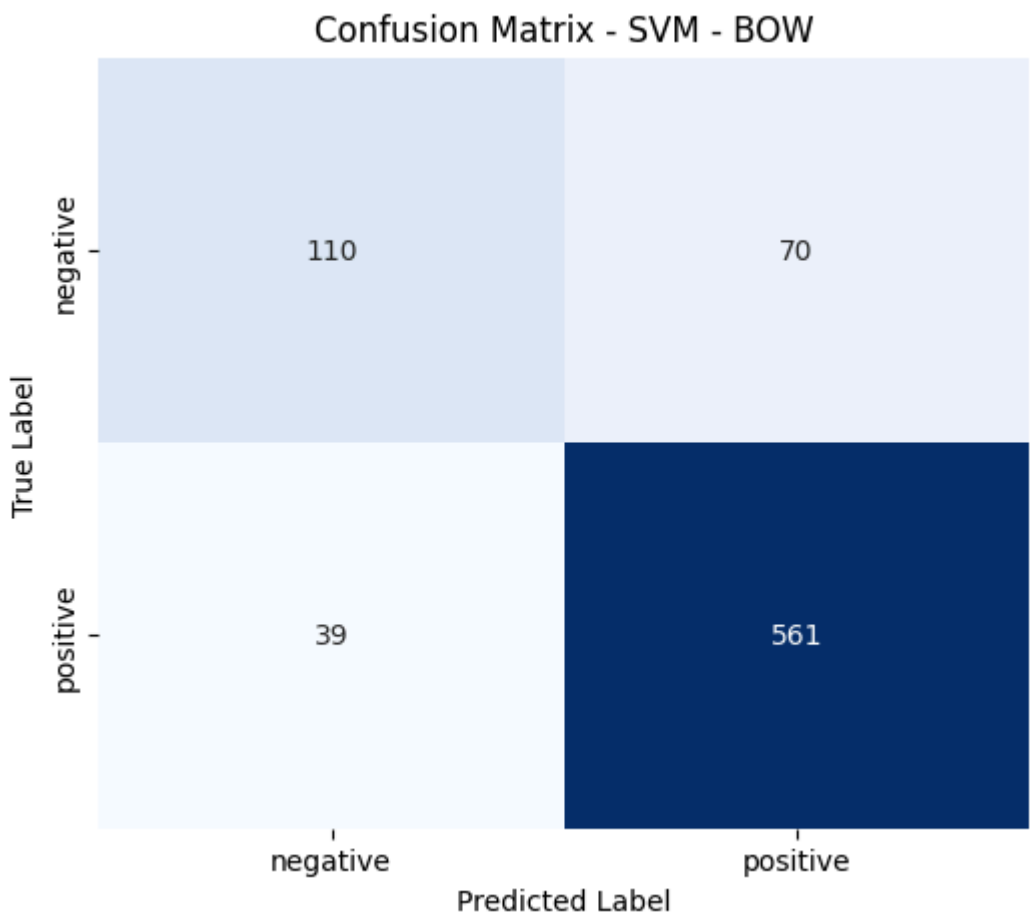
```
# CELL 46
# Note: Plot the confusion matrix for the best performing model.

cm = confusion_matrix(y_test, y_pred_best)

plt.figure(figsize=(6, 5))

sns.heatmap(
    cm,
    annot=True,
    fmt='d',
    cmap='Blues',
    cbar=False,
    xticklabels=np.unique(y_test),
    yticklabels=np.unique(y_test)
)

plt.title(f'Confusion Matrix - {best_model_name}')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

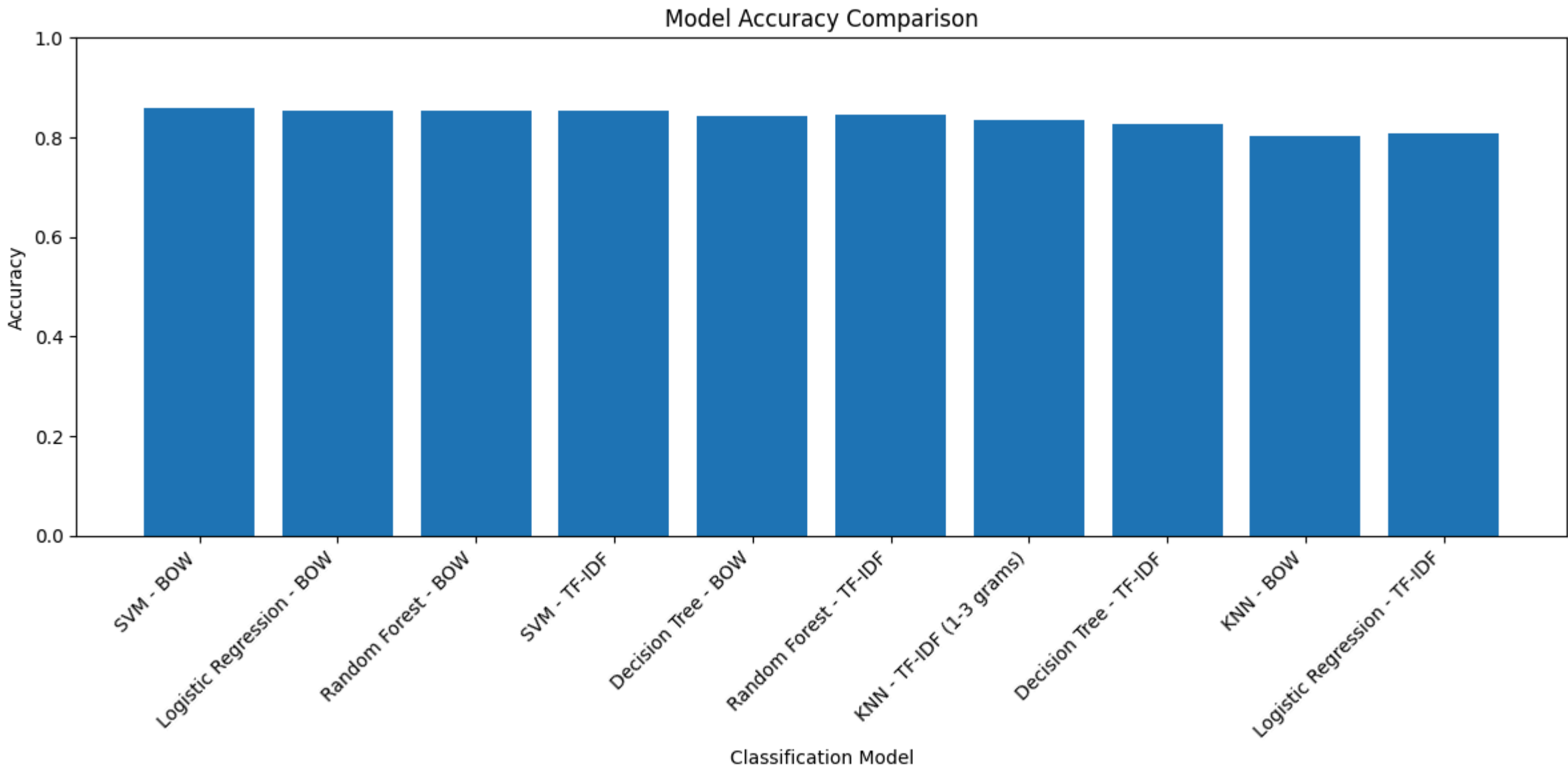


```
In [ ]: # CELL 47
# Note: Visualize the Accuracy of all classification models
# to compare their performance.

plt.figure(figsize=(12, 6))

plt.bar(
    final_results['Model'],
    final_results['Accuracy']
)

plt.title('Model Accuracy Comparison')
plt.xlabel('Classification Model')
plt.ylabel('Accuracy')
plt.xticks(rotation=45, ha='right')
plt.ylim(0, 1)
plt.tight_layout()
plt.show()
```

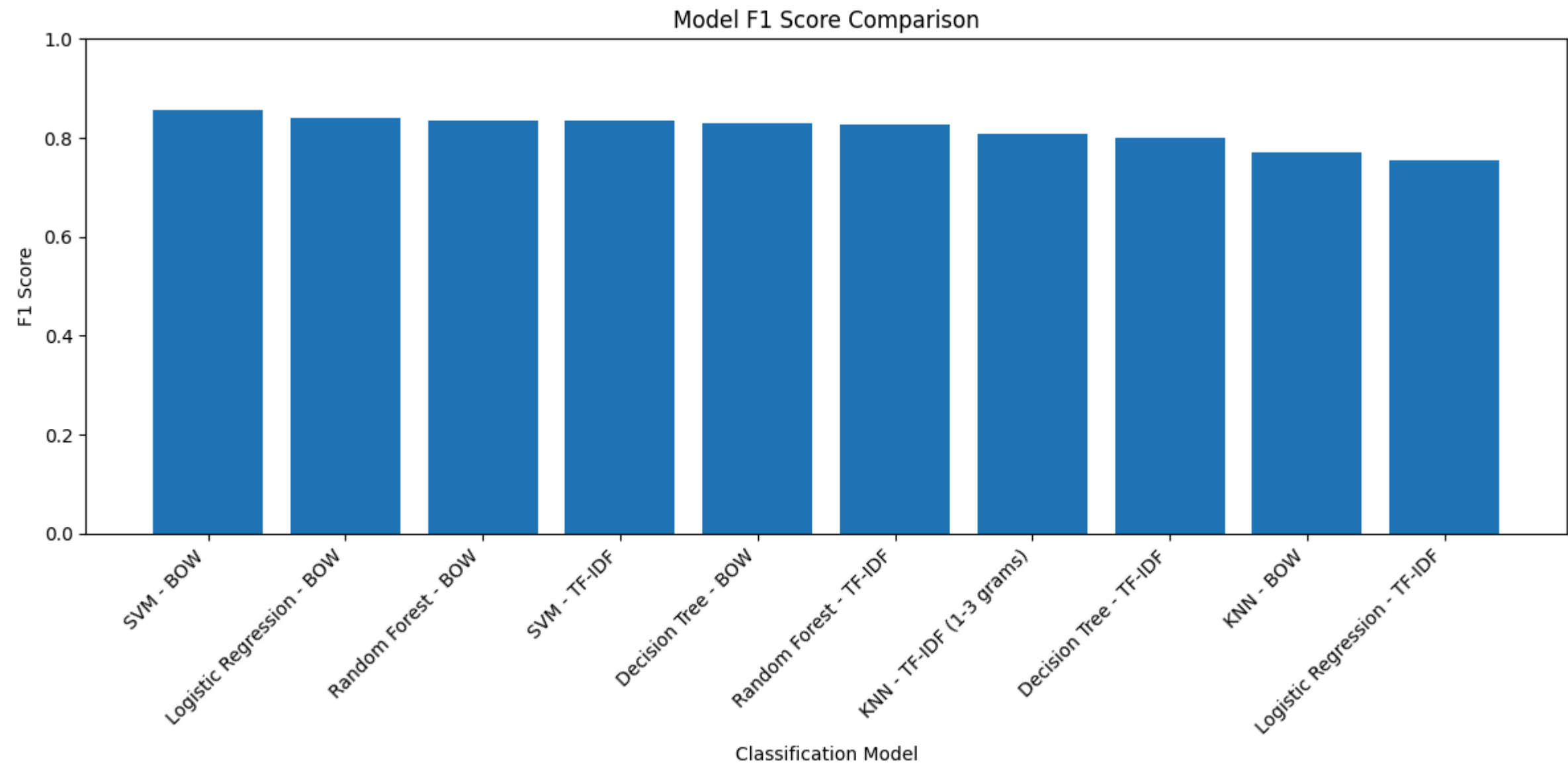


```
In [ ]: # CELL 48
# Note: Visualize the F1 Score of all classification models
# to compare their performance.

plt.figure(figsize=(12, 6))

plt.bar(
    final_results['Model'],
    final_results['F1 Score']
)

plt.title('Model F1 Score Comparison')
plt.xlabel('Classification Model')
plt.ylabel('F1 Score')
plt.xticks(rotation=45, ha='right')
plt.ylim(0, 1)
plt.tight_layout()
plt.show()
```

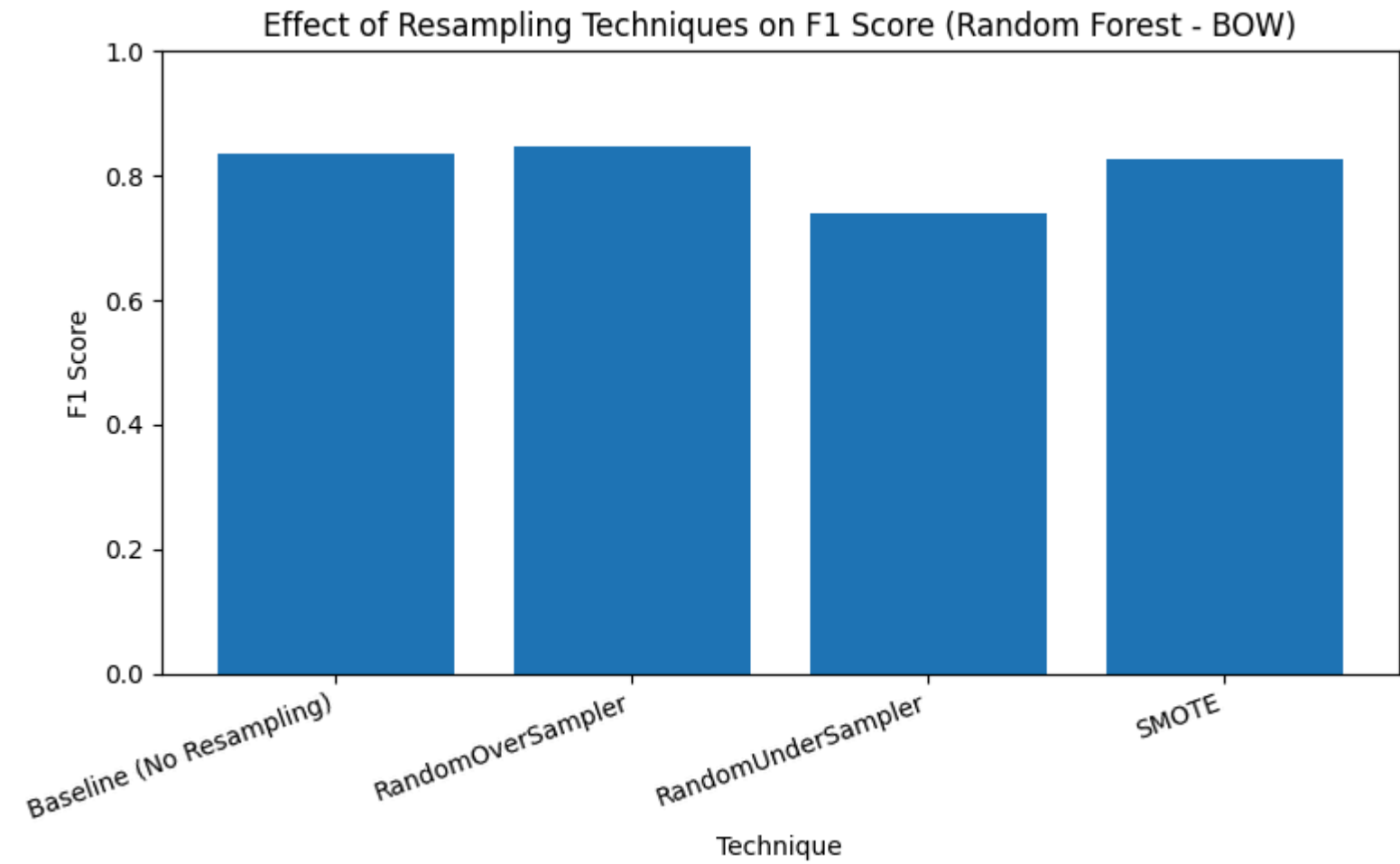


```
In [ ]: # CELL 49
# Note: Visualize the effect of each resampling technique on F1 Score,
# to clearly show how handling the imbalance changed performance.

plt.figure(figsize=(8, 5))

plt.bar(
    resampling_results['Technique'],
    resampling_results['F1 Score']
)

plt.title('Effect of Resampling Techniques on F1 Score (Random Forest - BOW)')
plt.xlabel('Technique')
plt.ylabel('F1 Score')
plt.xticks(rotation=20, ha='right')
plt.ylim(0, 1)
plt.tight_layout()
plt.show()
```



```
In [ ]: # CELL 50
# Note: Save the final model comparison results and the resampling
# comparison results as CSV files for future reference and submission.

final_results.reset_index(drop=True, inplace=True)

final_results.to_csv(
    "/content/drive/MyDrive/My Project 2/final_results.csv",
    index=False
)

resampling_results.to_csv(
    "/content/drive/MyDrive/My Project 2/resampling_results.csv",
    index=False
)

best_model_df = pd.DataFrame([best_model])

best_model_df.to_csv(
    "/content/drive/MyDrive/My Project 2/best_model_results.csv",
    index=False
)

print("Final results saved successfully.")
```

Final results saved successfully.

```
In [ ]: # CELL 51
# Note: Provide the final conclusion based on the comparison
# of all classification models and the resampling experiment.

print("Final Conclusion")
print("=====")

print(
    "The classification models were trained and evaluated "
    "using different feature representation methods (Bag of Words and TF-IDF)."
)

print(
    f"Based on the evaluation results, the best performing "
    f"model was {best_model_name}."
```

```
)

print(f"Accuracy: {best_model['Accuracy']:.4f}")
print(f"Precision: {best_model['Precision']:.4f}")
print(f"Recall: {best_model['Recall']:.4f}")
print(f"F1 Score: {best_model['F1 Score']:.4f}")

print(
    "\nThe dataset is imbalanced (2000 positive vs 600 negative samples). "
    "Resampling techniques (RandomOverSampler, RandomUnderSampler, and SMOTE) "
    "were applied and compared against the baseline Random Forest model to "
    "study their effect on the minority class performance."
)

best_resampling_row = resampling_results.sort_values(
    by='F1 Score', ascending=False
).iloc[0]

print(
    f"The best resampling technique was '{best_resampling_row['Technique']}' "
    f"with an F1 Score of {best_resampling_row['F1 Score']:.4f}."
)
```

Final Conclusion

=====

The classification models were trained and evaluated using different feature representation methods (Bag of Words and TF-IDF).

Based on the evaluation results, the best performing model was SVM - BOW.

Accuracy: 0.8603

Precision: 0.8543

Recall: 0.8603

F1 Score: 0.8554

The dataset is imbalanced (2000 positive vs 600 negative samples). Resampling techniques (RandomOverSampler, RandomUnderSampler, and SMOTE) were applied and compared against the baseline Random Forest model to study their effect on the minority class performance.

The best resampling technique was 'RandomOverSampler' with an F1 Score of 0.8457.

```
In [ ]: # CELL 52
# Note: Check that all the main steps required for the NLP
# classification task have been completed.

print("NLP Classification Project")
print("=====")

print("1. Dataset Download/Loading      : Completed")
print("2. Data Preprocessing             : Completed")
print("3. Dataset Splitting               : Completed")
print("4. Feature Representation          : Completed")
print("5. Model Training                  : Completed")
print("6. Model Evaluation                : Completed")
print("7. Model Comparison                : Completed")
print("8. Imbalanced Dataset Handling     : Completed")
print("9. Best Model Identification       : Completed")

print("\nBest Model:", best_model_name)
print("Best Accuracy:", round(best_model['Accuracy'], 4))
print("Best F1 Score:", round(best_model['F1 Score'], 4))
```

NLP Classification Project

=====

1. Dataset Download/Loading : Completed
2. Data Preprocessing : Completed
3. Dataset Splitting : Completed
4. Feature Representation : Completed
5. Model Training : Completed
6. Model Evaluation : Completed
7. Model Comparison : Completed
8. Imbalanced Dataset Handling : Completed
9. Best Model Identification : Completed

Best Model: SVM - BOW

Best Accuracy: 0.8603

Best F1 Score: 0.8554