



**Practical Image Processing and Natural Language Processing Second
Trimester 2026**

Project 2

Prepared By

Name: Sultan Ali Senan ID No.: 253002563

At

College of Informatics

Midocean University

Project 2 — Text Classification (Sentiment Analysis) using Machine Learning Models

Dataset: Unbalanced Sentiment Dataset (`unbalanceddataset.csv`) — 2,600 English texts labelled as *positive* or *negative*.

This notebook follows the steps required by the assignment:

1. Download the NLP dataset for the classification problem
2. Apply data preprocessing
3. Split the dataset
4. Apply feature representation methods
5. Train and evaluate models
6. Upload the code

Note on the dataset: the two classes are not equally represented (2,000 positive vs 600 negative, a ratio of about 3.3:1). An additional section on handling the class imbalance is therefore included at the end, following the techniques taught in Lecture 9.

Step 1 — Import Necessary Libraries

```
In [1]: import pandas as pd
import numpy as np
import re
import matplotlib.pyplot as plt
import seaborn as sns

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

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import LabelEncoder
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                             f1_score, confusion_matrix, classification_report)

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

import warnings
warnings.filterwarnings('ignore')

# Download required nltk resources (first run only)
nltk.download('punkt', quiet=True)
nltk.download('punkt_tab', quiet=True)
nltk.download('stopwords', quiet=True)

sns.set_style('whitegrid')
print("Libraries loaded")
```

Libraries loaded

Step 2 — Reading the Dataset

The dataset contains two columns: the raw text and its sentiment label.

```
In [2]: # File name as attached with the assignment
df = pd.read_csv("unbalanceddataset.csv", encoding="ISO-8859-1")

print("Shape:", df.shape)
df.head()
```

Shape: (2600, 2)

```
Out[2]:
```

	text	sentiment
0	Java Concurrency in Practice is probably the b...	positive
1	haha aww hun i bet you are more creative than me	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 [3]: df.info()
print()
print("Missing values:")
print(df.isnull().sum())
print()
print("Duplicated rows:", df.duplicated().sum())
```

```
<class 'pandas.DataFrame'>
RangeIndex: 2600 entries, 0 to 2599
Data columns (total 2 columns):
#   Column      Non-Null Count  Dtype
---  ---
0    text      2600 non-null    str
1    sentiment  2600 non-null    str
dtypes: str(2)
memory usage: 40.8 KB
```

```
Missing values:
text      0
sentiment 0
dtype: int64
```

```
Duplicated rows: 1
```

```
In [4]: # Class distribution
print(df['sentiment'].value_counts())
print()
print(df['sentiment'].value_counts(normalize=True).round(3))
```

```
sentiment
positive    2000
negative     600
Name: count, dtype: int64
```

```
sentiment
positive    0.769
negative    0.231
Name: proportion, dtype: float64
```

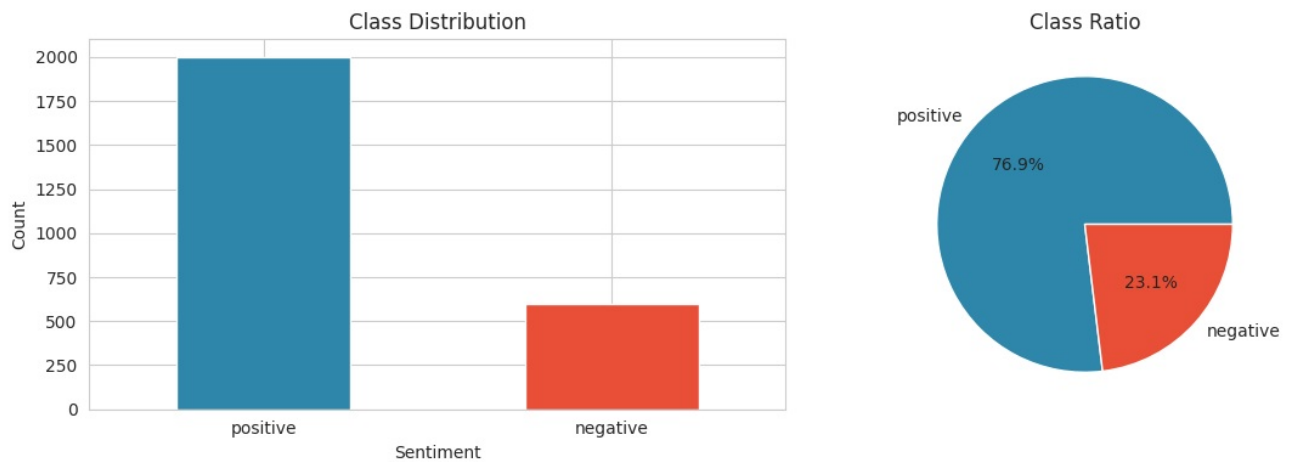
The *positive* class represents about 77% of the data against 23% for the *negative* class, i.e. an imbalance ratio close to 3:1. This affects model behaviour and is addressed in the final section.

```
In [5]: fig, ax = plt.subplots(1, 2, figsize=(12, 4))
```

```
df['sentiment'].value_counts().plot(kind='bar', ax=ax[0], color=['#2E86AB', '#E94F37'])
ax[0].set_title('Class Distribution')
ax[0].set_xlabel('Sentiment')
ax[0].set_ylabel('Count')
ax[0].tick_params(axis='x', rotation=0)

df['sentiment'].value_counts().plot(kind='pie', ax=ax[1], autopct='%1.1f%%',
                                     colors=['#2E86AB', '#E94F37'])
ax[1].set_ylabel('')
ax[1].set_title('Class Ratio')

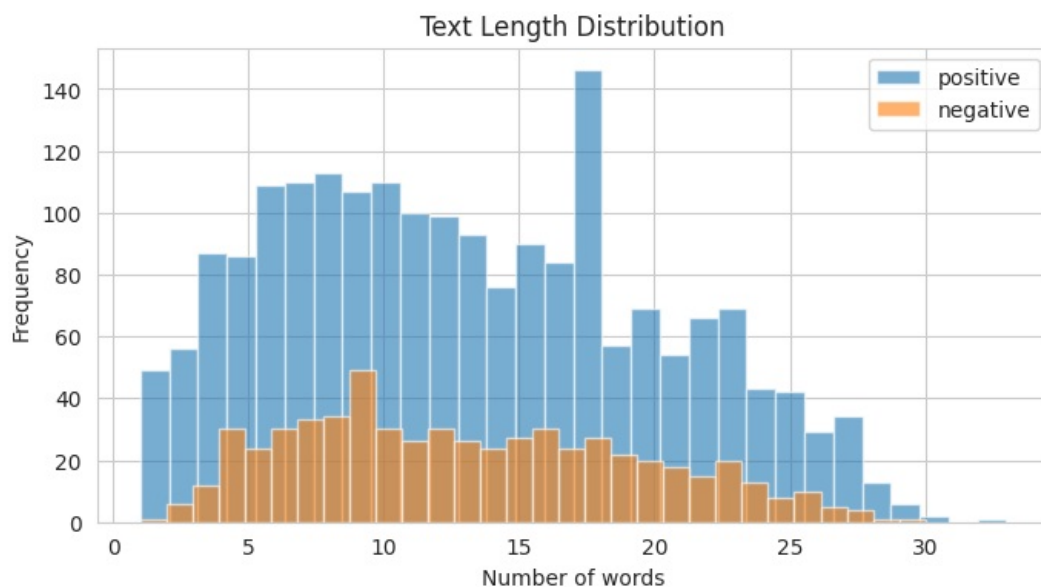
plt.tight_layout()
plt.show()
```



```
In [6]: # Text length before preprocessing
df['word_count'] = df['text'].astype(str).apply(lambda x: len(x.split()))
print(df.groupby('sentiment')['word_count'].describe().round(2))
```

```
plt.figure(figsize=(8, 4))
for s in df['sentiment'].unique():
    plt.hist(df[df['sentiment'] == s]['word_count'], bins=30, alpha=0.6, label=s)
plt.xlabel('Number of words')
plt.ylabel('Frequency')
plt.title('Text Length Distribution')
plt.legend()
plt.show()
```

	count	mean	std	min	25%	50%	75%	max
negative	600.0	13.13	6.48	1.0	8.0	12.0	18.0	30.0
positive	2000.0	13.12	6.83	1.0	8.0	12.0	18.0	33.0



Step 3 — Applying Data Preprocessing

The following steps are applied to every text:

- Lowercasing
- Removing URLs and HTML tags
- Removing numbers and punctuation
- Tokenization
- Removing stop words
- Stemming using `PorterStemmer`

```
In [7]: stemmer = PorterStemmer()
stop_words = set(stopwords.words('english'))

def preprocess_text(text):
    text = str(text).lower()
    text = re.sub(r'http\S+|www\S+', '', text)      # URLs
    text = re.sub(r'<.*?>', '', text)             # HTML tags
    text = re.sub(r'^a-zA-Z\s', '', text)          # numbers and symbols
    tokens = word_tokenize(text)
    tokens = [w for w in tokens if w not in stop_words and len(w) > 2]
    tokens = [stemmer.stem(w) for w in tokens]
    return ' '.join(tokens)

df['clean_text'] = df['text'].apply(preprocess_text)
df = df[df['clean_text'].str.strip() != '']        # drop rows that became empty after cleaning
df = df.dropna().reset_index(drop=True)

print("Shape after cleaning:", df.shape)
```

Shape after cleaning: (2597, 4)

```
In [8]: # Compare text before and after preprocessing
for i in range(3):
    print("Before:", df['text'][i][:90])
    print("After :", df['clean_text'][i][:90])
    print("-" * 70)
```

Before: Java Concurrency in Practice is probably the best Java book I've ever bought. There's a re
After : java concurr practic probabl best java book ive ever bought there recip interrupt block op

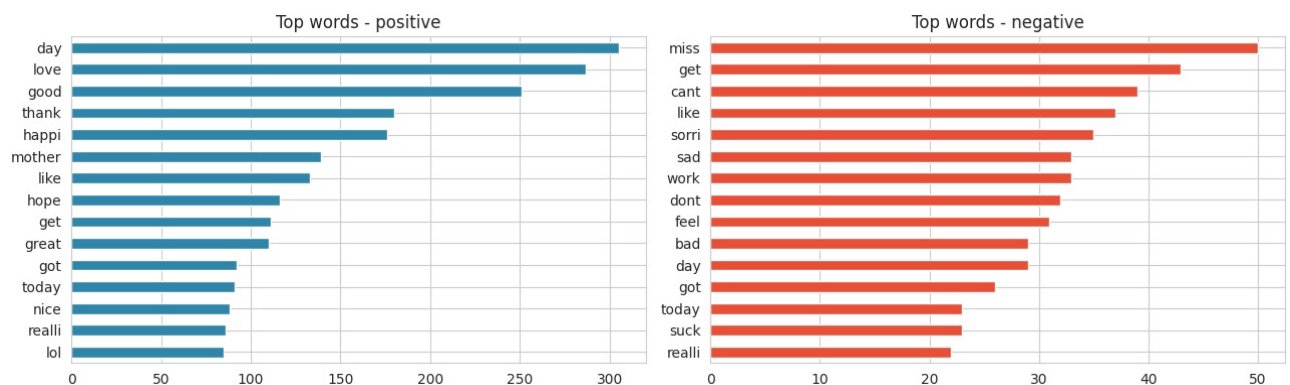
Before: haha aww hun i bet you are more creative than me
After : haha aww hun bet creativ

Before: _pickle lol, thank you very much Hope you're having a great day!
After : pickl lol thank much hope your great day

```
In [9]: # Most frequent words per class
fig, ax = plt.subplots(1, 2, figsize=(13, 4))

for i, s in enumerate(['positive', 'negative']):
    cv = CountVectorizer(max_features=15)
    counts = cv.fit_transform(df[df['sentiment'] == s]['clean_text'])
    freq = pd.Series(counts.toarray().sum(axis=0), index=cv.get_feature_names_out())
    freq.sort_values().plot(kind='barh', ax=ax[i], color='#2E86AB' if i == 0 else '#E94F37')
    ax[i].set_title(f'Top words - {s}')

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



Step 4 — Splitting the Dataset

The data is split into 70% training and 30% testing. `stratify` is used so that both sets keep the same class proportions, which matters because the dataset is imbalanced.

```
In [10]: X = df['clean_text']

# Label encoding: negative=0 , positive=1
le = LabelEncoder()
y = le.fit_transform(df['sentiment'])
print("Classes:", dict(zip(le.classes_, le.transform(le.classes_))))
```

```
Classes: {'negative': np.int64(0), 'positive': np.int64(1)}
```

```
In [11]: X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=42, stratify=y)

print("Train size:", X_train.shape[0])
print("Test size :", X_test.shape[0])
print()
print("Train distribution:", Counter(y_train))
print("Test distribution :", Counter(y_test))
```

```
Train size: 1817
Test size : 780
```

```
Train distribution: Counter({np.int64(1): 1398, np.int64(0): 419})
Test distribution : Counter({np.int64(1): 600, np.int64(0): 180})
```

Step 5 — Feature Representation

Three representation methods are applied and compared:

Method	Description
Bag of Words	Raw count of each word
TF-IDF Unigrams	Word weighted by its importance in the document against the corpus
TF-IDF Unigrams + Bigrams	Same as above, with word pairs added to capture context

Important: the vectorizer is fitted on the training set only and then applied to the test set, in order to avoid data leakage.

```
In [12]: representations = {}

# Bag of Words
bow = CountVectorizer()
representations['Bow'] = (bow.fit_transform(X_train), bow.transform(X_test))

# TF-IDF unigrams
tfidf1 = TfidfVectorizer(ngram_range=(1, 1))
representations['TF-IDF (1,1)'] = (tfidf1.fit_transform(X_train), tfidf1.transform(X_test))

# TF-IDF unigrams + bigrams
tfidf2 = TfidfVectorizer(ngram_range=(1, 2))
representations['TF-IDF (1,2)'] = (tfidf2.fit_transform(X_train), tfidf2.transform(X_test))

for name, (tr, te) in representations.items():
    print(f"{name:15s} -> train {tr.shape}, test {te.shape}")
```

```
BoW          -> train (1817, 3560), test (780, 3560)
TF-IDF (1,1) -> train (1817, 3560), test (780, 3560)
TF-IDF (1,2) -> train (1817, 13330), test (780, 13330)
```

```
In [13]: # Sample of the extracted features
print("Sample features (TF-IDF 1,2):")
print(list(tfidf2.get_feature_names_out()[3000:3010]))
```

```
Sample features (TF-IDF 1,2):
['ebm', 'ebru', 'ebru thank', 'econom', 'econom downturn', 'economy', 'eddi', 'eddi brenden', 'eddi izzard', 'edinburgh']
```

Step 6 — Training and Evaluating the Models

Six models are trained on each representation, with hyperparameters tuned using `GridSearchCV`.

The main evaluation metric is **F1-macro** rather than accuracy, because the dataset is imbalanced and accuracy alone can be misleading.

```
In [14]: def evaluate_model(model_name, y_true, y_pred, show_cm=False):
    metrics = {
        'Model': model_name,
        '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 (weighted)': f1_score(y_true, y_pred, average='weighted'),
        'F1 (macro)': f1_score(y_true, y_pred, average='macro'),
    }

    if show_cm:
        cm = confusion_matrix(y_true, y_pred)
        plt.figure(figsize=(4, 3.5))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
                    xticklabels=le.classes_, yticklabels=le.classes_)
        plt.title(f'Confusion Matrix - {model_name}')
        plt.xlabel('Predicted')
        plt.ylabel('True')
        plt.show()
        print(classification_report(y_true, y_pred, target_names=le.classes_))

    return metrics
```

```
In [15]: # Models and their hyperparameter grids
models = {
    'LogisticRegression': (LogisticRegression(random_state=42, max_iter=1000),
                          {'C': [0.1, 1, 10], 'solver': ['liblinear', 'lbfgs']}),

    'DecisionTree': (DecisionTreeClassifier(random_state=42),
                    {'criterion': ['gini', 'entropy'], 'max_depth': [None, 20, 40]}),

    'RandomForest': (RandomForestClassifier(random_state=42),
                    {'n_estimators': [100, 200, 300], 'max_depth': [None, 20]}),

    'SVM': (SVC(random_state=42),
            {'kernel': ['linear', 'rbf'], 'C': [1, 10]}),

    'KNN': (KNeighborsClassifier(),
            {'n_neighbors': [3, 5, 7], 'weights': ['uniform', 'distance']}),

    'NaiveBayes': (MultinomialNB(),
                  {'alpha': [0.1, 0.5, 1.0]}),
}
```

```

In [16]: results = []
best_estimators = {}

for rep_name, (Xtr, Xte) in representations.items():
    print(f"\n===== {rep_name} =====")
    for model_name, (estimator, grid) in models.items():
        gs = GridSearchCV(estimator, grid, cv=5, scoring='f1_macro', n_jobs=-1)
        gs.fit(Xtr, y_train)
        y_pred = gs.best_estimator_.predict(Xte)

        m = evaluate_model(model_name, y_test, y_pred)
        m['Representation'] = rep_name
        m['Best Params'] = gs.best_params_
        results.append(m)
        best_estimators[(rep_name, model_name)] = gs.best_estimator_

    print(f"{model_name:20s} Acc={m['Accuracy']:.4f} F1-macro={m['F1 (macro)']:.4f}")

```

```

===== BoW =====
LogisticRegression Acc=0.8436 F1-macro=0.7666
DecisionTree Acc=0.8397 F1-macro=0.7702
RandomForest Acc=0.8526 F1-macro=0.7462
SVM Acc=0.8436 F1-macro=0.7725
KNN Acc=0.8179 F1-macro=0.7070
NaiveBayes Acc=0.8564 F1-macro=0.7615

```

```

===== TF-IDF (1,1) =====
LogisticRegression Acc=0.8513 F1-macro=0.7606
DecisionTree Acc=0.8192 F1-macro=0.7408
RandomForest Acc=0.8538 F1-macro=0.7417
SVM Acc=0.8359 F1-macro=0.7583
KNN Acc=0.8385 F1-macro=0.7145
NaiveBayes Acc=0.8397 F1-macro=0.7142

```

```

===== TF-IDF (1,2) =====
LogisticRegression Acc=0.8551 F1-macro=0.7520
DecisionTree Acc=0.8462 F1-macro=0.7762
RandomForest Acc=0.8526 F1-macro=0.7402
SVM Acc=0.8615 F1-macro=0.7867
KNN Acc=0.8410 F1-macro=0.7441
NaiveBayes Acc=0.8397 F1-macro=0.6953

```

```

In [17]: results_df = pd.DataFrame(results)[
    ['Representation', 'Model', 'Accuracy', 'Precision', 'Recall',
     'F1 (weighted)', 'F1 (macro)', 'Best Params']]

results_df.sort_values('F1 (macro)', ascending=False).round(4).head(10)

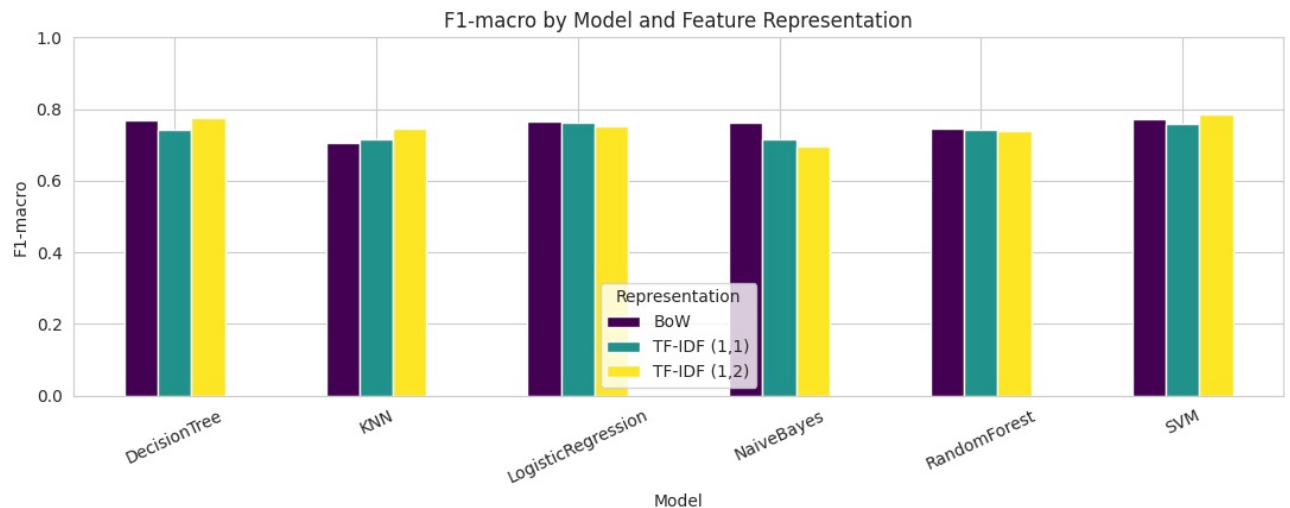
```

	Representation	Model	Accuracy	Precision	Recall	F1 (weighted)	F1 (macro)	Best Params
15	TF-IDF (1,2)	SVM	0.8615	0.8550	0.8615	0.8547	0.7867	{'C': 10, 'kernel': 'linear'}
13	TF-IDF (1,2)	DecisionTree	0.8462	0.8419	0.8462	0.8436	0.7762	{'criterion': 'entropy', 'max_depth': None}
3	BoW	SVM	0.8436	0.8393	0.8436	0.8410	0.7725	{'C': 1, 'kernel': 'linear'}
1	BoW	DecisionTree	0.8397	0.8371	0.8397	0.8383	0.7702	{'criterion': 'gini', 'max_depth': None}
0	BoW	LogisticRegression	0.8436	0.8368	0.8436	0.8388	0.7666	{'C': 10, 'solver': 'liblinear'}
5	BoW	NaiveBayes	0.8564	0.8520	0.8564	0.8425	0.7615	{'alpha': 0.5}
6	TF-IDF (1,1)	LogisticRegression	0.8513	0.8436	0.8513	0.8400	0.7606	{'C': 10, 'solver': 'lbfgs'}
9	TF-IDF (1,1)	SVM	0.8359	0.8298	0.8359	0.8320	0.7583	{'C': 10, 'kernel': 'linear'}
12	TF-IDF (1,2)	LogisticRegression	0.8551	0.8539	0.8551	0.8381	0.7520	{'C': 10, 'solver': 'lbfgs'}
2	BoW	RandomForest	0.8526	0.8513	0.8526	0.8347	0.7462	{'max_depth': None, 'n_estimators': 100}

```
In [18]: # Comparing models across the three representations
pivot = results_df.pivot(index='Model', columns='Representation', values='F1 (macro)')

pivot.plot(kind='bar', figsize=(11, 4.5), colormap='viridis')
plt.title('F1-macro by Model and Feature Representation')
plt.ylabel('F1-macro')
plt.xticks(rotation=25)
plt.legend(title='Representation')
plt.ylim(0, 1)
plt.tight_layout()
plt.show()

print(pivot.round(4))
```



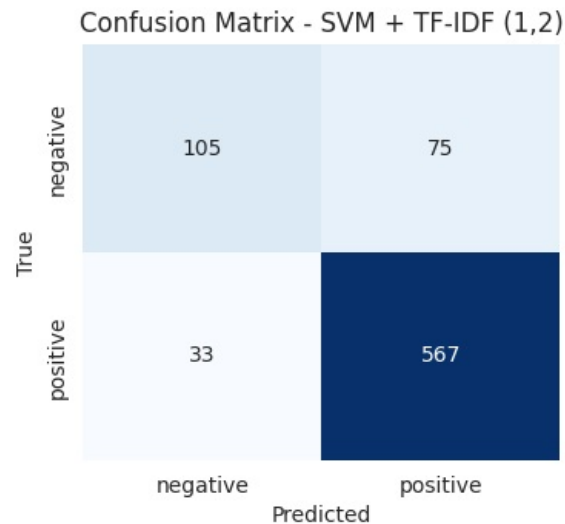
Representation	BoW	TF-IDF (1,1)	TF-IDF (1,2)
Model			
DecisionTree	0.7702	0.7408	0.7762
KNN	0.7070	0.7145	0.7441
LogisticRegression	0.7666	0.7606	0.7520
NaiveBayes	0.7615	0.7142	0.6953
RandomForest	0.7462	0.7417	0.7402
SVM	0.7725	0.7583	0.7867

```
In [19]: # Best model
best_row = results_df.loc[results_df['F1 (macro)'].idxmax()]
print("Best model      :", best_row['Model'])
print("Representation  :", best_row['Representation'])
print("Best params     :", best_row['Best Params'])
print("Accuracy        : %.4f" % best_row['Accuracy'])
print("F1-macro         : %.4f" % best_row['F1 (macro)'])
```

```
Best model      : SVM
Representation  : TF-IDF (1,2)
Best params     : {'C': 10, 'kernel': 'linear'}
Accuracy        : 0.8615
F1-macro       : 0.7867
```

```
In [20]: # Detailed performance of the best model
key = (best_row['Representation'], best_row['Model'])
Xtr_b, Xte_b = representations[best_row['Representation']]
y_pred_best = best_estimators[key].predict(Xte_b)

_ = evaluate_model(f"{best_row['Model']} + {best_row['Representation']}",
                  y_test, y_pred_best, show_cm=True)
```



	precision	recall	f1-score	support
negative	0.76	0.58	0.66	180
positive	0.88	0.94	0.91	600
accuracy			0.86	780
macro avg	0.82	0.76	0.79	780
weighted avg	0.85	0.86	0.85	780

Observation: the confusion matrix shows that the model identifies the *positive* class well, while the recall of the *negative* class stays relatively low. The model was trained on data dominated by positive samples, so it became biased towards the majority class. This problem is addressed in the following section.

Step 7 — Handling the Imbalanced Dataset

The following techniques are applied and compared:

- **RandomOverSampler** — randomly duplicates samples of the minority class
- **RandomUnderSampler** — randomly removes samples of the majority class
- **SMOTE** — generates synthetic minority samples by interpolating between nearest neighbours
- **class_weight='balanced'** — gives a higher weight to the minority class inside the model itself

Note: these techniques are applied to the training set only. The test set is left untouched so that the evaluation remains realistic.

```
In [21]: # Use the best representation from the previous section
Xtr_bal, Xte_bal = representations[best_row['Representation']]
print("Using representation:", best_row['Representation'])
print("Original training distribution:", Counter(y_train))
```

```
Using representation: TF-IDF (1,2)
Original training distribution: Counter({np.int64(1): 1398, np.int64(0): 419})
```

```
In [22]: samplers = {
    'RandomOverSampler': RandomOverSampler(random_state=42),
    'RandomUnderSampler': RandomUnderSampler(random_state=42),
    'SMOTE': SMOTE(random_state=42),
}

resampled = {'Original': (Xtr_bal, y_train)}

for name, sampler in samplers.items():
    Xr, yr = sampler.fit_resample(Xtr_bal, y_train)
    resampled[name] = (Xr, yr)
    print(f"{name:20s} -> {Counter(yr)}")
```

```
RandomOverSampler -> Counter({np.int64(1): 1398, np.int64(0): 1398})
RandomUnderSampler -> Counter({np.int64(0): 419, np.int64(1): 419})
SMOTE -> Counter({np.int64(1): 1398, np.int64(0): 1398})
```

```
In [23]: balance_results = []

for name, (Xr, yr) in resampled.items():
    for model_name in ['LogisticRegression', 'RandomForest', 'SVM']:
        estimator, grid = models[model_name]
        gs = GridSearchCV(estimator, grid, cv=5, scoring='f1_macro', n_jobs=-1)
        gs.fit(Xr, yr)
        y_pred = gs.best_estimator_.predict(Xte_bal)

        m = evaluate_model(model_name, y_test, y_pred)
        m['Method'] = name
        balance_results.append(m)
    print(f"{name:20s} {model_name:20s} Acc={m['Accuracy']:.4f} F1-macro={m['F1 (macro)']:.4f}")
```

Original	LogisticRegression	Acc=0.8551	F1-macro=0.7520
Original	RandomForest	Acc=0.8526	F1-macro=0.7402
Original	SVM	Acc=0.8615	F1-macro=0.7867
RandomOverSampler	LogisticRegression	Acc=0.8603	F1-macro=0.7946
RandomOverSampler	RandomForest	Acc=0.8641	F1-macro=0.7835
RandomOverSampler	SVM	Acc=0.8321	F1-macro=0.6764
RandomUnderSampler	LogisticRegression	Acc=0.8128	F1-macro=0.7744
RandomUnderSampler	RandomForest	Acc=0.7718	F1-macro=0.7346
RandomUnderSampler	SVM	Acc=0.7987	F1-macro=0.7595
SMOTE	LogisticRegression	Acc=0.8603	F1-macro=0.7901
SMOTE	RandomForest	Acc=0.8526	F1-macro=0.7417
SMOTE	SVM	Acc=0.8615	F1-macro=0.7867

```
In [24]: # Fourth technique: adjusting class weights inside the model
for model_name, estimator, grid in [
    ('LogisticRegression', LogisticRegression(random_state=42, max_iter=1000,
                                                class_weight='balanced')),
    ({'C': [0.1, 1, 10], 'solver': ['liblinear', 'lbfgs']}),
    ('RandomForest', RandomForestClassifier(random_state=42, class_weight='balanced'),
     {'n_estimators': [100, 200, 300], 'max_depth': [None, 20]}),
    ('SVM', SVC(random_state=42, class_weight='balanced'),
     {'kernel': ['linear', 'rbf'], 'C': [1, 10]}),
]:
    gs = GridSearchCV(estimator, grid, cv=5, scoring='f1_macro', n_jobs=-1)
    gs.fit(Xtr_bal, y_train)
    y_pred = gs.best_estimator_.predict(Xte_bal)

    m = evaluate_model(model_name, y_test, y_pred)
    m['Method'] = 'class_weight'
    balance_results.append(m)
    print(f"{'class_weight':20s} {model_name:20s} Acc={m['Accuracy']:.4f} F1-macro={m['F1 (macro)']:.4f}")
```

class_weight	LogisticRegression	Acc=0.8538	F1-macro=0.8090
class_weight	RandomForest	Acc=0.8564	F1-macro=0.7903
class_weight	SVM	Acc=0.8526	F1-macro=0.7833

```
In [25]: balance_df = pd.DataFrame(balance_results)[
        ['Method', 'Model', 'Accuracy', 'Precision', 'Recall', 'F1 (weighted)', 'F1 (macro)']]

balance_df.sort_values('F1 (macro)', ascending=False).round(4)
```

```
Out[25]:
```

	Method	Model	Accuracy	Precision	Recall	F1 (weighted)	F1 (macro)
12	class_weight	LogisticRegression	0.8538	0.8693	0.8538	0.8588	0.8090
3	RandomOverSampler	LogisticRegression	0.8603	0.8556	0.8603	0.8571	0.7946
13	class_weight	RandomForest	0.8564	0.8521	0.8564	0.8537	0.7903
9	SMOTE	LogisticRegression	0.8603	0.8543	0.8603	0.8554	0.7901
11	SMOTE	SVM	0.8615	0.8550	0.8615	0.8547	0.7867
2	Original	SVM	0.8615	0.8550	0.8615	0.8547	0.7867
4	RandomOverSampler	RandomForest	0.8641	0.8585	0.8641	0.8546	0.7835
14	class_weight	SVM	0.8526	0.8476	0.8526	0.8493	0.7833
6	RandomUnderSampler	LogisticRegression	0.8128	0.8620	0.8128	0.8245	0.7744
8	RandomUnderSampler	SVM	0.7987	0.8535	0.7987	0.8118	0.7595
0	Original	LogisticRegression	0.8551	0.8539	0.8551	0.8381	0.7520
10	SMOTE	RandomForest	0.8526	0.8542	0.8526	0.8328	0.7417
1	Original	RandomForest	0.8526	0.8554	0.8526	0.8322	0.7402
7	RandomUnderSampler	RandomForest	0.7718	0.8446	0.7718	0.7881	0.7346
5	RandomOverSampler	SVM	0.8321	0.8460	0.8321	0.7972	0.6764

```
In [26]: pivot_bal = balance_df.pivot(index='Method', columns='Model', values='F1 (macro)')
order = ['Original', 'RandomOverSampler', 'RandomUnderSampler', 'SMOTE', 'class_weight']
pivot_bal = pivot_bal.reindex(order)

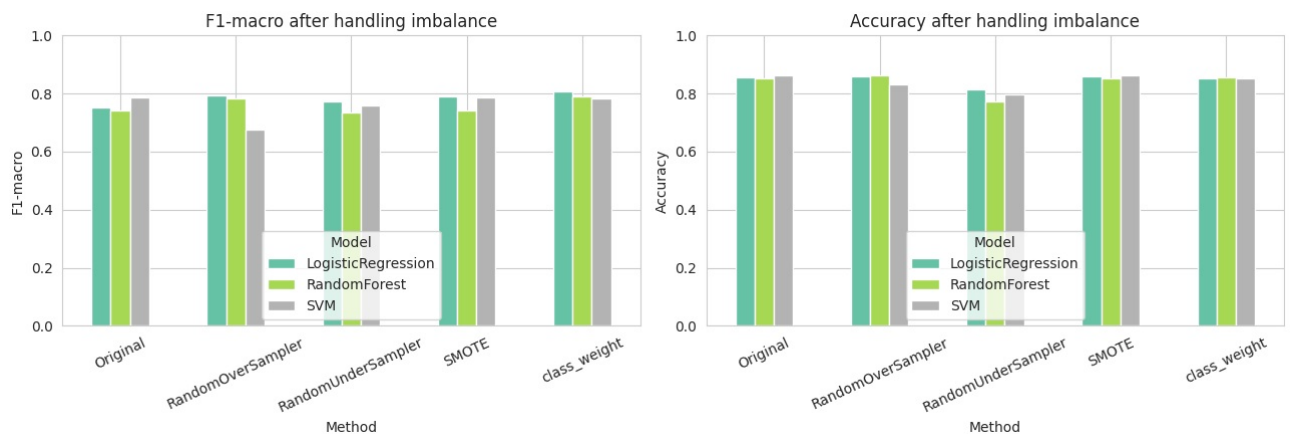
fig, ax = plt.subplots(1, 2, figsize=(13, 4.5))

pivot_bal.plot(kind='bar', ax=ax[0], colormap='Set2')
ax[0].set_title('F1-macro after handling imbalance')
ax[0].set_ylabel('F1-macro')
ax[0].tick_params(axis='x', rotation=25)
ax[0].set_ylim(0, 1)

pivot_acc = balance_df.pivot(index='Method', columns='Model', values='Accuracy').reindex(order)
pivot_acc.plot(kind='bar', ax=ax[1], colormap='Set2')
ax[1].set_title('Accuracy after handling imbalance')
ax[1].set_ylabel('Accuracy')
ax[1].tick_params(axis='x', rotation=25)
ax[1].set_ylim(0, 1)

plt.tight_layout()
plt.show()

print(pivot_bal.round(4))
```



Model	LogisticRegression	RandomForest	SVM
Method			
Original	0.7520	0.7402	0.7867
RandomOverSampler	0.7946	0.7835	0.6764
RandomUnderSampler	0.7744	0.7346	0.7595
SMOTE	0.7901	0.7417	0.7867
class_weight	0.8090	0.7903	0.7833

```
In [27]: # Best result after handling the imbalance
best_bal = balance_df.loc[balance_df['F1 (macro)'].idxmax()]
print("Best method :", best_bal['Method'])
print("Best model  :", best_bal['Model'])
print("Accuracy    : %.4f" % best_bal['Accuracy'])
print("F1-macro    : %.4f" % best_bal['F1 (macro)'])
print()
print("Improvement in F1-macro vs original data: %.4f" %
      (best_bal['F1 (macro)'] - balance_df[balance_df['Method'] == 'Original']['F1 (macro)'].max()))

Best method : class_weight
Best model  : LogisticRegression
Accuracy    : 0.8538
F1-macro    : 0.8090

Improvement in F1-macro vs original data: 0.0223
```

Conclusion

In this project we read the sentiment dataset (2,600 texts), explored its class distribution and found it imbalanced with a ratio of about 3:1 in favour of the positive class. The texts were cleaned (lowercasing, URL/HTML/punctuation removal, tokenization, stop-word removal and stemming), then split into 70% training and 30% testing using stratified sampling. Three feature representations were built (Bag of Words, TF-IDF unigrams, TF-IDF unigrams + bigrams) and six models were trained on each of them with `GridSearchCV`, giving 18 experiments in total. Finally, four techniques for handling class imbalance were applied and compared.

Results summary:

Stage	Best combination	Accuracy	F1-macro
Before handling imbalance	SVM + TF-IDF (1,2)	0.8615	0.7867
After handling imbalance	LogisticRegression + <code>class_weight</code>	0.8538	0.8090

Main observations:

- No single representation is best in every case. Bag of Words gave the best result with LogisticRegression and NaiveBayes, while TF-IDF with bigrams was the best for SVM and achieved the highest overall score. The choice of representation is therefore tied to the model used.
- Accuracy on its own is a misleading metric here: a model that always predicts *positive* reaches about 77% accuracy without learning anything. F1-macro was used instead, both for comparison and for hyperparameter tuning.
- Before handling the imbalance, the recall of the negative class did not exceed 0.58 despite the high overall accuracy — clear evidence of bias towards the majority class.
- Adjusting the class weights (`class_weight='balanced'`) performed best, improving F1-macro by 0.022 with only a small drop in accuracy. This is an acceptable trade-off since the goal is to improve the detection of the minority class.
- RandomOverSampler and SMOTE gave smaller improvements, while RandomUnderSampler was the weakest because it discarded more than 900 training samples.