

```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that gets preserved as output when you create a new notebook.
# You can also write temporary files to /kaggle/temp/, but they won't be saved outside of the current session
```

Imports

```
import os
import re
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.naive_bayes import MultinomialNB
from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import (
    accuracy_score,
    precision_recall_fscore_support,
    classification_report,
    confusion_matrix,
    roc_auc_score
)
```

Load dataset

```
DATA_DIR = "/kaggle/input/project2-dataset"

print("Files in dataset folder:")
print(os.listdir(DATA_DIR))

# If the filename differs, pick it from the printed list
DATA_PATH = os.path.join(DATA_DIR, "unbalanceddataset (1).csv")

df = pd.read_csv(DATA_PATH)

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

```
Files in dataset folder:
['unbalanceddataset (1).csv']
Shape: (2600, 2)
Columns: ['text', 'sentiment']
```

	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

Quick check + label encoding

```
# Keep only required columns and drop missing values
df = df.dropna(subset=["text", "sentiment"]).copy()

print("Class distribution:")
print(df["sentiment"].value_counts())

# Encode labels
label_map = {"negative": 0, "positive": 1}
df["y"] = df["sentiment"].map(label_map)

# Safety check
assert df["y"].isna().sum() == 0, "Unmapped labels found. Check df['sentiment'].unique()."
print("Unique labels:", df["sentiment"].unique())
```

```
Class distribution:
sentiment
positive    2000
negative     600
Name: count, dtype: int64
Unique labels: ['positive' 'negative']
```

Apply data preprocessing

```
def clean_text(text: str) -> str:
    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"@|_|#", " ", text)              # mentions/hashtags
    text = re.sub(r"[^a-z\s]", " ", text)            # keep letters + spaces
    text = re.sub(r"\s+", " ", text).strip()         # normalize spaces
    return text

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

df[["text", "text_clean", "sentiment"]].head()
```

	text	text_clean	sentiment
0	Java Concurrency in Practice is probably the b...	java concurrency in practice is probably the b...	positive
1	haha aww hun i bet you are more creative tha...	haha aww hun i bet you are more creative than me	positive
2	_pickle lol, thank you very much Hope you`re h...	pickle lol thank you very much hope you re hav...	positive
3	Out for an evening on the town with jeremy. Sa...	out for an evening on the town with jeremy sad...	negative
4	- just took over the #1 Most Endorsed spot on...	just took over the most endorsed spot on twind...	positive

Split the dataset

```
X = df["text_clean"].values
y = df["y"].values

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.20,
    random_state=42,
    stratify=y
)

print("Train size:", len(X_train), "Test size:", len(X_test))
```

```
Train size: 2080 Test size: 520
```

Feature representation methods (TF-IDF)

```
# Word-level TF-IDF
tfidf_word = TfidfVectorizer(ngram_range=(1,2), max_features=5000)
X_train_word = tfidf_word.fit_transform(X_train)
```

```

X_test_word = tfidf_word.transform(X_test)

# Character-level TF-IDF
tfidf_char = TfidfVectorizer(analyzer="char", ngram_range=(3,5), max_features=8000)
X_train_char = tfidf_char.fit_transform(X_train)
X_test_char = tfidf_char.transform(X_test)

print("Word TF-IDF shape:", X_train_word.shape)
print("Char TF-IDF shape:", X_train_char.shape)

```

```

Word TF-IDF shape: (2080, 5000)
Char TF-IDF shape: (2080, 8000)

```

✓ Train and evaluate models

```

def evaluate_model(name, model, Xtr, Xte, ytr, yte):
    model.fit(Xtr, ytr)
    pred = model.predict(Xte)

    acc = accuracy_score(yte, pred)
    p, r, f1, _ = precision_recall_fscore_support(yte, pred, average="binary", zero_division=0)

    auc = None
    if hasattr(model, "predict_proba"):
        proba = model.predict_proba(Xte)[:, 1]
        auc = roc_auc_score(yte, proba)
    elif hasattr(model, "decision_function"):
        scores = model.decision_function(Xte)
        auc = roc_auc_score(yte, scores)

    print(f"*80")
    print(name)
    if auc is not None:
        print(f"Accuracy: {acc:.4f} | Precision: {p:.4f} | Recall: {r:.4f} | F1: {f1:.4f} | ROC-AUC: {auc:.4f}")
    else:
        print(f"Accuracy: {acc:.4f} | Precision: {p:.4f} | Recall: {r:.4f} | F1: {f1:.4f}")

    print("\nClassification Report:\n",
          classification_report(yte, pred, target_names=["negative", "positive"], zero_division=0))
    print("Confusion Matrix:\n", confusion_matrix(yte, pred))
    return model, f1, auc if auc is not None else -1

results = []

# Word TF-IDF models
lr_model, lr_f1, lr_auc = evaluate_model(
    "Logistic Regression (Word TF-IDF)",
    LogisticRegression(max_iter=2000, class_weight="balanced"),
    X_train_word, X_test_word, y_train, y_test
)
results.append(("Logistic Regression (Word TF-IDF)", lr_f1, lr_auc, lr_model))

svm_word_model, svm_word_f1, svm_word_auc = evaluate_model(
    "Linear SVM (Word TF-IDF)",
    LinearSVC(class_weight="balanced"),
    X_train_word, X_test_word, y_train, y_test
)
results.append(("Linear SVM (Word TF-IDF)", svm_word_f1, svm_word_auc, svm_word_model))

nb_model, nb_f1, nb_auc = evaluate_model(
    "Multinomial Naive Bayes (Word TF-IDF)",
    MultinomialNB(),
    X_train_word, X_test_word, y_train, y_test
)
results.append(("Multinomial Naive Bayes (Word TF-IDF)", nb_f1, nb_auc, nb_model))

rf_model, rf_f1, rf_auc = evaluate_model(
    "Random Forest (Word TF-IDF)",
    RandomForestClassifier(n_estimators=300, random_state=42, class_weight="balanced"),
    X_train_word, X_test_word, y_train, y_test
)
results.append(("Random Forest (Word TF-IDF)", rf_f1, rf_auc, rf_model))

# Char TF-IDF model
svm_char_model, svm_char_f1, svm_char_auc = evaluate_model(
    "Linear SVM (Char TF-IDF)",
    LinearSVC(class_weight="balanced"),
    X_train_char, X_test_char, y_train, y_test
)

```

```
)
results.append(("Linear SVM (Char TF-IDF)", svm_char_f1, svm_char_auc, svm_char_model))
```

```

accuracy          0.83    520
macro avg         0.76    0.76    0.76    520
weighted avg      0.83    0.83    0.83    520
```

Confusion Matrix:

```
[[ 75  45]
 [ 43 357]]
```

=====

Multinomial Naive Bayes (Word TF-IDF)

Accuracy: 0.7808 | Precision: 0.7782 | Recall: 1.0000 | F1: 0.8753 | ROC-AUC: 0.8804

Classification Report:

```

              precision    recall  f1-score   support

negative      1.00        0.05        0.10         120
positive       0.78        1.00        0.88         400
```

```

accuracy          0.78    520
macro avg         0.89    0.53    0.49    520
weighted avg      0.83    0.78    0.70    520
```

Confusion Matrix:

```
[[  6 114]
 [  0 400]]
```

=====

Random Forest (Word TF-IDF)

Accuracy: 0.8250 | Precision: 0.8381 | Recall: 0.9575 | F1: 0.8938 | ROC-AUC: 0.8680

Classification Report:

```

              precision    recall  f1-score   support

negative       0.73        0.38        0.50         120
positive       0.84        0.96        0.89         400
```

```

accuracy          0.82    520
macro avg         0.78    0.67    0.70    520
weighted avg      0.81    0.82    0.80    520
```

Confusion Matrix:

```
[[ 46  74]
 [ 17 383]]
```

=====

Linear SVM (Char TF-IDF)

Accuracy: 0.8346 | Precision: 0.8925 | Recall: 0.8925 | F1: 0.8925 | ROC-AUC: 0.8793

Classification Report:

```

              precision    recall  f1-score   support

negative       0.64        0.64        0.64         120
positive       0.89        0.89        0.89         400
```

```

accuracy          0.83    520
macro avg         0.77    0.77    0.77    520
weighted avg      0.83    0.83    0.83    520
```

Confusion Matrix:

```
[[ 77  43]
 [ 43 357]]
```

▼ Save best model + vectorizer

```
import joblib

# Pick best by F1 (requirement-safe)
best = sorted(results, key=lambda x: x[1], reverse=True)[0]
best_name, best_f1, best_auc, best_model = best

print("Best model:", best_name, "| F1:", best_f1, "| AUC:", best_auc)

# Save matching vectorizer
if "Char" in best_name:
    joblib.dump(tfidf_char, "tfidf_best.pkl")
else:
    joblib.dump(tfidf_word, "tfidf_best.pkl")

joblib.dump(best_model, "best_model.pkl")

print("Saved: tfidf_best.pkl and best_model.pkl in /kaggle/working")
```

```
Best model: Random Forest (Word TF-IDF) | F1: 0.8938156359393232 | AUC: 0.8680312499999999  
Saved: tfidf_best.pkl and best_model.pkl in /kaggle/working
```