

project2

August 22, 2026

1 Project 2 — Sentiment Classification (Lectures 6–10) — unbalanceddataset.csv (2 classes, imbalanced)

Pipeline stages:

1. **Data preprocessing** — cleaning, tokenization, stopword removal, stemming
2. **Dataset splitting** — stratified 60 / 20 / 20 train / validation / test
3. **Feature representation** — BoW, TF-IDF, TF-IDF n-grams, Word2Vec CBOW, Word2Vec Skip-gram
4. **Model training & evaluation** — Logistic Regression, Decision Tree, Random Forest, KNN, SVM, Naive Bayes

Datasets: `unbalanceddataset.csv` (raw text, 2 classes, imbalanced) and `clean_dataset.csv` (3 classes).

1.1 Import necessary libraries

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

from nlp_pipeline import (
    ensure_nltk_resources, clean_text, preprocess_text, preprocess_dataframe,
    split_dataset, build_features, train_and_evaluate, evaluate_model,
    sentence_vector, RANDOM_STATE,
)

ensure_nltk_resources()
pd.set_option("display.width", 160)
sns.set_theme(style="whitegrid")
np.random.seed(RANDOM_STATE)
```

1.2 Reading the dataset

`unbalanceddataset.csv` holds raw, uncleaned text, so it exercises every preprocessing step. It is imbalanced 2000 positive / 600 negative, so `class_weight="balanced"` is used.

Swap the two assignment lines below to run this same notebook on the other dataset.

```
[2]: DATASET, TEXT_COL, LABEL_COL, CLASS_WEIGHT = "unbalanceddataset.csv", "text",
      ↪ "sentiment", "balanced"
      # DATASET, TEXT_COL, LABEL_COL, CLASS_WEIGHT = "clean_dataset.csv",
      ↪ "clean_text", "sentiment", None

      df = pd.read_csv(DATASET, encoding="ISO-8859-1")
      print(df.shape)
      df.head()
```

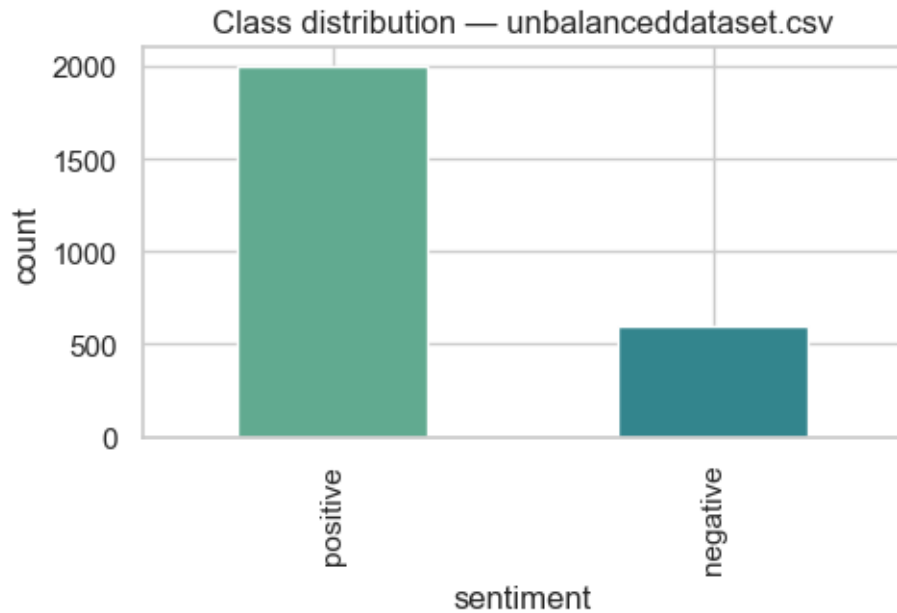
(2600, 2)

```
[2]:                                     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
```

```
[3]: df[LABEL_COL].value_counts()
```

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

```
[4]: ax = df[LABEL_COL].value_counts().plot(kind="bar", color=sns.
      ↪ color_palette("crest", 3), figsize=(5, 3.5))
      ax.set_title(f"Class distribution - {DATASET}")
      ax.set_ylabel("count")
      plt.tight_layout(); plt.show()
```



1.3 Step 1 — Data preprocessing (Lecture 6)

`clean_text` lowercases and strips HTML, URLs, emails, mentions, numbers and punctuation. `preprocess_text` then tokenizes, drops English stopwords, and applies the Porter stemmer.

```
[5]: sample = "Check http://example.com - I LOVED this Java book!! Bought 3 copies_
↳@amazon <b>great</b>"
print("raw      :", sample)
print("cleaned  :", clean_text(sample))
print("tokens   :", preprocess_text(sample))
```

```
raw      : Check http://example.com - I LOVED this Java book!! Bought 3 copies
@amazon <b>great</b>
cleaned  : check i loved this java book bought copies great
tokens   : ['check', 'love', 'java', 'book', 'bought', 'copi', 'great']
```

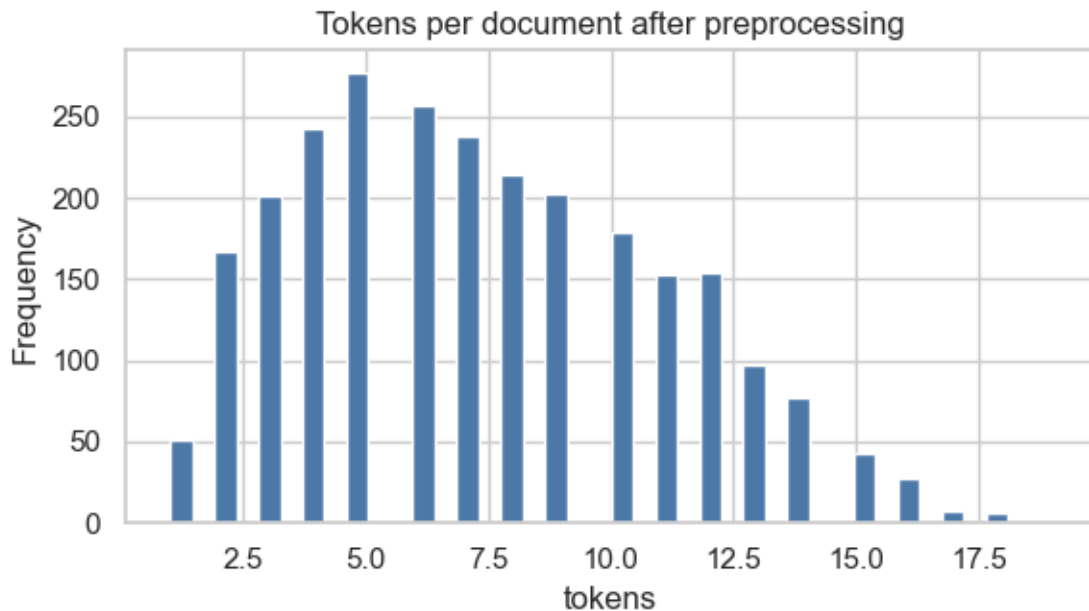
```
[6]: df = preprocess_dataframe(df, TEXT_COL, LABEL_COL, method="stem")
df[[TEXT_COL, "processed_text", LABEL_COL]].head()
```

```
rows: 2600 -> 2600 (dedup/na) -> 2598 (non-empty after cleaning)
avg tokens/doc: 7.4
```

```
[6]:                                     text
processed_text sentiment
0  Java Concurrency in Practice is probably the b...  java concurr practic
probabl best java book iv...  positive
1  haha aww hun i bet you are more creative tha...
```

haha aww hun bet creativ	positive	
2 _pickle lol, thank you very much Hope you`re h...		pickl lol thank
much hope your great day	positive	
3 Out for an evening on the town with jeremy. Sa...		even town
jeremi sad carri cant come	negative	
4 - just took over the #1 Most Endorsed spot on...		took endors spot
twindexxcom thank endors	positive	

```
[7]: # how long are the documents after cleaning?
lengths = df["tokens"].map(len)
ax = lengths.plot(kind="hist", bins=40, figsize=(6, 3.5), color="#4C78A8")
ax.set_title("Tokens per document after preprocessing"); ax.set_xlabel("tokens")
plt.tight_layout(); plt.show()
lengths.describe()
```



```
[7]: count    2598.000000
      mean       7.353734
      std       3.688064
      min       1.000000
      25%       4.000000
      50%       7.000000
      75%      10.000000
      max      19.000000
      Name: tokens, dtype: float64
```

1.4 Step 2 — Splitting the dataset (Lecture 7)

Stratified split keeps the class ratio identical in every part, which matters for the imbalanced data. The validation set is held out for model selection; the test set is only touched at the end.

```
[8]: train_df, val_df, test_df = split_dataset(df, LABEL_COL, test_size=0.2,
      ↪ val_size=0.2, stratify=True)
y_train, y_val, y_test = train_df[LABEL_COL].values, val_df[LABEL_COL].values,
      ↪ test_df[LABEL_COL].values
len(train_df), len(val_df), len(test_df)
```

```
train n=1558    class %: {'positive': 77.0, 'negative': 23.0}
val    n=520    class %: {'positive': 76.9, 'negative': 23.1}
test   n=520    class %: {'positive': 76.9, 'negative': 23.1}
```

```
[8]: (1558, 520, 520)
```

1.5 Step 3 — Feature representation (Lectures 7–10)

Five representations, two families:

Family	Representation	Notes
Count-based	Bag of Words	CountVectorizer, raw term counts
Count-based	TF-IDF	down-weights common terms
Count-based	TF-IDF (1,2)-gram	adds bigrams, captures short phrases
Prediction-based	Word2Vec CBOW (sg=0)	predicts the target word from context
Prediction-based	Word2Vec Skip-gram (sg=1)	predicts context from the target word

Word2Vec produces one vector per *word*, so each document is represented by the **average** of its word vectors — a fixed-size dense vector usable by any sklearn classifier.

Every vectorizer and both Word2Vec models are fit on the **training split only**, so no test information leaks into the feature space.

```
[9]: reps = build_features(train_df, test_df, max_features=5000, w2v_size=100)
```

```
BoW          train shape (1558, 3285) (sparse)
TF-IDF       train shape (1558, 3285) (sparse)
TF-IDF (1,2)-gram train shape (1558, 5000) (sparse)
Word2Vec-CBOW train shape (1558, 100) (dense)
Word2Vec-SkipGram train shape (1558, 100) (dense)
Word2Vec vocab: CBOW 1178, SkipGram 1178
```

```
[10]: # what the count-based features look like
bow = reps["BoW"]["extractor"]
```

```

print("BoW vocabulary size:", len(bow.vocabulary_))
print("first 20 features   :", bow.get_feature_names_out()[:20])

tfidf2 = reps["TF-IDF (1,2)-gram"]["extractor"]
bigrams = [f for f in tfidf2.get_feature_names_out() if " " in f]
print("\nbigrams kept:", len(bigrams), "->", bigrams[:10])

```

BoW vocabulary size: 3285

```

first 20 features   : ['aaaaw' 'aaah' 'aaand' 'abbi' 'abil' 'abl' 'abo'
'absolut' 'absolutli'
'abt' 'abundantbless' 'accident' 'accidentali' 'accidentcholla'
'accomplish' 'account' 'ace' 'ach' 'achiev' 'acid']

```

```

bigrams kept: 3155 -> ['aaaaw window', 'aaah sweater', 'aaand stoke', 'abbi
accident', 'abil move', 'abl achiev', 'abl breath', 'abl get', 'abl squeez',
'act fool']

```

```

[11]: # what the Word2Vec embeddings learned
cbow = reps["Word2Vec-CBOW"]["extractor"]
sg    = reps["Word2Vec-SkipGram"]["extractor"]
print("CBOW vocab:", len(cbow.wv), " SkipGram vocab:", len(sg.wv))

for word in ["good", "bad", "love"]:
    if word in sg.wv:
        near = [w for w, _ in sg.wv.most_similar(word, topn=5)]
        print(f" most similar to {word!r}: {near}")

```

CBOW vocab: 1178 SkipGram vocab: 1178

```

most similar to 'good': ['im', 'way', 'got', 'like', 'love']
most similar to 'bad':  ['love', 'like', 'get', 'new', 'good']
most similar to 'love': ['im', 'got', 'come', 'cant', 'think']

```

1.6 Step 4 — Training and evaluating the models (Lectures 7–10)

Six classifiers per representation. Naive Bayes switches between MultinomialNB (sparse counts) and GaussianNB (dense Word2Vec vectors), since multinomial NB requires non-negative features. On the imbalanced dataset every model that supports it uses `class_weight="balanced"`.

Both **weighted** and **macro** F1 are reported: weighted F1 is dominated by the majority class, so macro F1 is the honest score on imbalanced data and is used for ranking.

```

[12]: large = len(train_df) > 6000    # swap SVC(rbf) for LinearSVC on the big dataset
results, details = train_and_evaluate(reps, y_train, y_test,
                                     class_weight=CLASS_WEIGHT, large=large)

```

--- BoW ---

LogisticRegression	acc=0.8519	weighted-F1=0.8559	macro-F1=0.8031
DecisionTree	acc=0.8058	weighted-F1=0.8153	macro-F1=0.7559

RandomForest	acc=0.8596	weighted-F1=0.8424	macro-F1=0.7580
KNN	acc=0.8058	weighted-F1=0.7714	macro-F1=0.6398
SVM (RBF)	acc=0.8519	weighted-F1=0.8538	macro-F1=0.7967
NaiveBayes	acc=0.8481	weighted-F1=0.8256	macro-F1=0.7286
--- TF-IDF ---			
LogisticRegression	acc=0.8558	weighted-F1=0.8586	macro-F1=0.8052
DecisionTree	acc=0.7904	weighted-F1=0.8000	macro-F1=0.7342
RandomForest	acc=0.8519	weighted-F1=0.8364	macro-F1=0.7511
KNN	acc=0.8250	weighted-F1=0.8046	macro-F1=0.7008
SVM (RBF)	acc=0.8577	weighted-F1=0.8447	macro-F1=0.7655
NaiveBayes	acc=0.7846	weighted-F1=0.7036	macro-F1=0.5011
--- TF-IDF (1,2)-gram ---			
LogisticRegression	acc=0.8577	weighted-F1=0.8617	macro-F1=0.8112
DecisionTree	acc=0.7923	weighted-F1=0.7990	macro-F1=0.7272
RandomForest	acc=0.8577	weighted-F1=0.8447	macro-F1=0.7655
KNN	acc=0.8154	weighted-F1=0.7933	macro-F1=0.6831
SVM (RBF)	acc=0.8442	weighted-F1=0.8287	macro-F1=0.7403
NaiveBayes	acc=0.7846	weighted-F1=0.7036	macro-F1=0.5011
--- Word2Vec-CBOW ---			
LogisticRegression	acc=0.6173	weighted-F1=0.6451	macro-F1=0.5576
DecisionTree	acc=0.6904	weighted-F1=0.6908	macro-F1=0.5652
RandomForest	acc=0.7635	weighted-F1=0.6823	macro-F1=0.4698
KNN	acc=0.7788	weighted-F1=0.7345	macro-F1=0.5774
SVM (RBF)	acc=0.7346	weighted-F1=0.7507	macro-F1=0.6780
NaiveBayes	acc=0.6731	weighted-F1=0.6917	macro-F1=0.5986
--- Word2Vec-SkipGram ---			
LogisticRegression	acc=0.6250	weighted-F1=0.6517	macro-F1=0.5636
DecisionTree	acc=0.6577	weighted-F1=0.6632	macro-F1=0.5339
RandomForest	acc=0.7673	weighted-F1=0.6845	macro-F1=0.4716
KNN	acc=0.7808	weighted-F1=0.7302	macro-F1=0.5655
SVM (RBF)	acc=0.6385	weighted-F1=0.6616	macro-F1=0.5664
NaiveBayes	acc=0.6385	weighted-F1=0.6608	macro-F1=0.5631

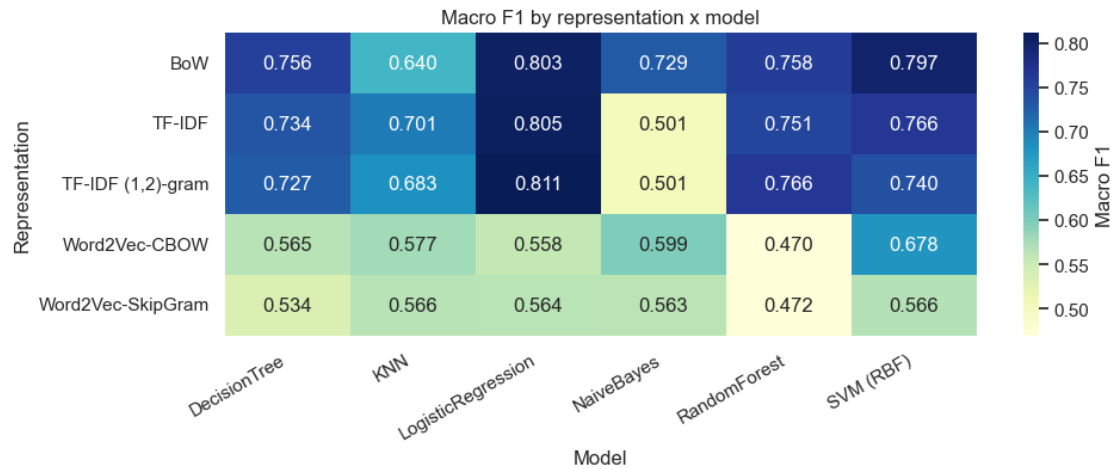
```
[13]: results.head(15)
```

```
[13]:      Representation      Model  Accuracy  Precision  Recall  F1
Score  Macro F1
0  TF-IDF (1,2)-gram  LogisticRegression  0.857692  0.869083  0.857692
```

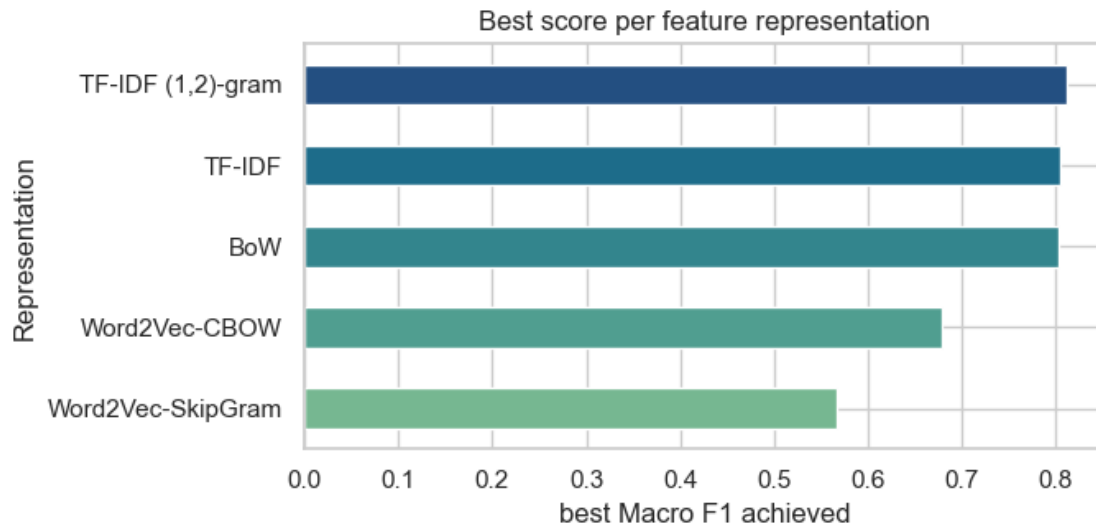
0.861655	0.811221				
1	TF-IDF	LogisticRegression	0.855769	0.863137	0.855769
0.858633	0.805170				
2	BoW	LogisticRegression	0.851923	0.862974	0.851923
0.855884	0.803068				
3	BoW	SVM (RBF)	0.851923	0.856187	0.851923
0.853757	0.796709				
4	TF-IDF	SVM (RBF)	0.857692	0.852857	0.857692
0.844690	0.765543				
5	TF-IDF (1,2)-gram	RandomForest	0.857692	0.852857	0.857692
0.844690	0.765543				
6	BoW	RandomForest	0.859615	0.861287	0.859615
0.842421	0.757956				
7	BoW	DecisionTree	0.805769	0.837230	0.805769
0.815315	0.755920				
8	TF-IDF	RandomForest	0.851923	0.847363	0.851923
0.836382	0.751055				
9	TF-IDF (1,2)-gram	SVM (RBF)	0.844231	0.836637	0.844231
0.828745	0.740255				
10	TF-IDF	DecisionTree	0.790385	0.819647	0.790385
0.800015	0.734245				
11	BoW	NaiveBayes	0.848077	0.851523	0.848077
0.825571	0.728621				
12	TF-IDF (1,2)-gram	DecisionTree	0.792308	0.809772	0.792308
0.798970	0.727225				
13	TF-IDF	KNN	0.825000	0.812680	0.825000
0.804615	0.700838				
14	TF-IDF (1,2)-gram	KNN	0.815385	0.799679	0.815385
0.793330	0.683056				

1.6.1 Comparing representations and models

```
[14]: pivot = results.pivot(index="Representation", columns="Model", values="Macro_
      ↪F1")
      plt.figure(figsize=(10, 4.2))
      sns.heatmap(pivot, annot=True, fmt=".3f", cmap="YlGnBu", cbar_kws={"label":_
      ↪"Macro F1"})
      plt.title("Macro F1 by representation x model"); plt.xticks(rotation=30,_
      ↪ha="right")
      plt.tight_layout(); plt.show()
```



```
[15]: best_rep = results.groupby("Representation")["Macro F1"].max().sort_values()
ax = best_rep.plot(kind="barh", figsize=(7, 3.5), color=sns.
    ↳color_palette("crest", len(best_rep)))
ax.set_xlabel("best Macro F1 achieved"); ax.set_title("Best score per feature_
    ↳representation")
plt.tight_layout(); plt.show()
```



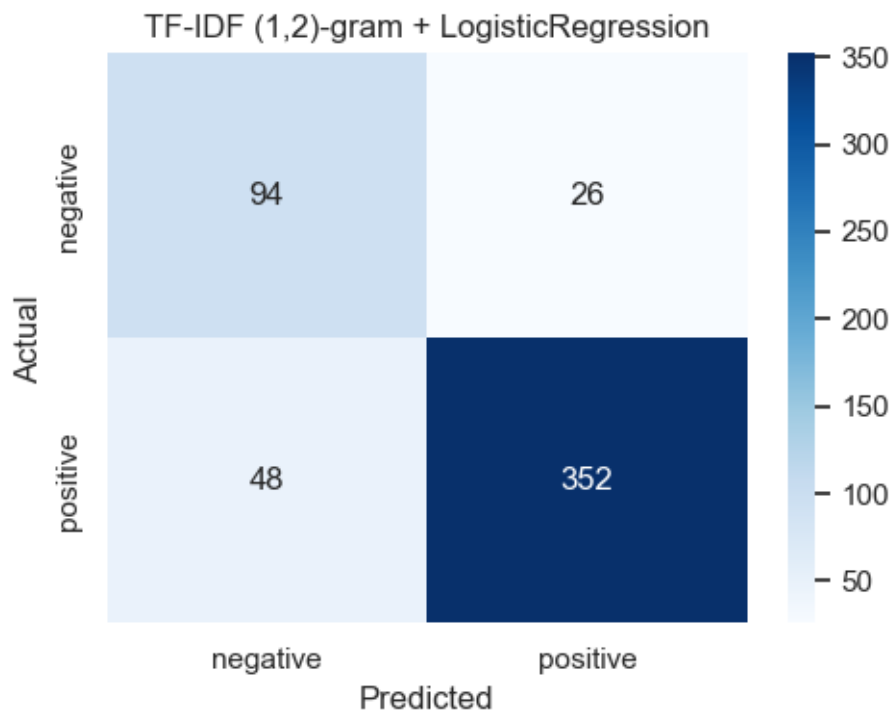
1.6.2 Best model — full report and confusion matrix

```
[16]: best = results.iloc[0]
res = details[(best["Representation"], best["Model"])]
print(f"BEST: {best['Representation']} + {best['Model']}")
print(f"Accuracy {res['Accuracy']:.4f} | weighted-F1 {res['F1 Score']:.4f} |
      ↳macro-F1 {res['Macro F1']:.4f}\n")
print(res["Classification Report"])
```

BEST: TF-IDF (1,2)-gram + LogisticRegression
Accuracy 0.8577 | weighted-F1 0.8617 | macro-F1 0.8112

	precision	recall	f1-score	support
negative	0.66	0.78	0.72	120
positive	0.93	0.88	0.90	400
accuracy			0.86	520
macro avg	0.80	0.83	0.81	520
weighted avg	0.87	0.86	0.86	520

```
[17]: labels = sorted(pd.unique(y_test))
plt.figure(figsize=(5, 4))
sns.heatmap(res["Confusion Matrix"], annot=True, fmt="d", cmap="Blues",
            xticklabels=labels, yticklabels=labels)
plt.xlabel("Predicted"); plt.ylabel("Actual")
plt.title(f"{best['Representation']} + {best['Model']}")
plt.tight_layout(); plt.show()
```



1.7 Conclusions

- **Count-based features beat averaged Word2Vec** on both datasets. With only a few thousand training documents there is not enough text to learn good embeddings from scratch, and averaging word vectors discards word order entirely. Pretrained embeddings would be the fix.
- **TF-IDF with bigrams + Logistic Regression** wins on the imbalanced dataset; **BoW + Random Forest** wins on the 3-class clean dataset.
- **class_weight="balanced" matters**: without it the models collapse toward the majority class and macro F1 drops sharply even though accuracy looks fine — which is exactly why macro F1 is the ranking metric here.
- The **neutral class** is the hard one in the 3-class task (F1 0.42 vs 0.80 for positive); neutral text simply carries fewer discriminative sentiment words.

Run `python3 run_pipeline.py` to reproduce both datasets end to end and write everything to `results/`.