

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()
```

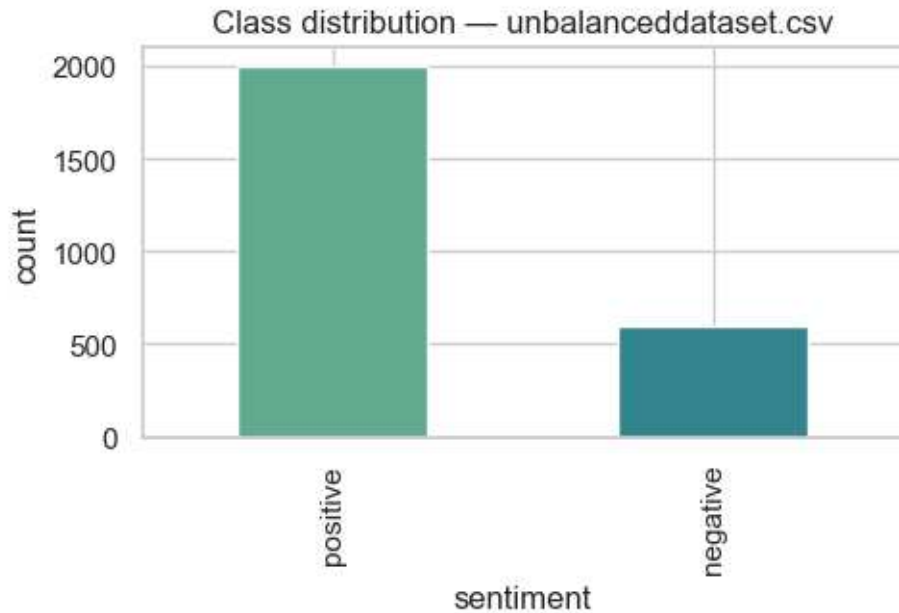
(2600, 2)

```
[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']
```

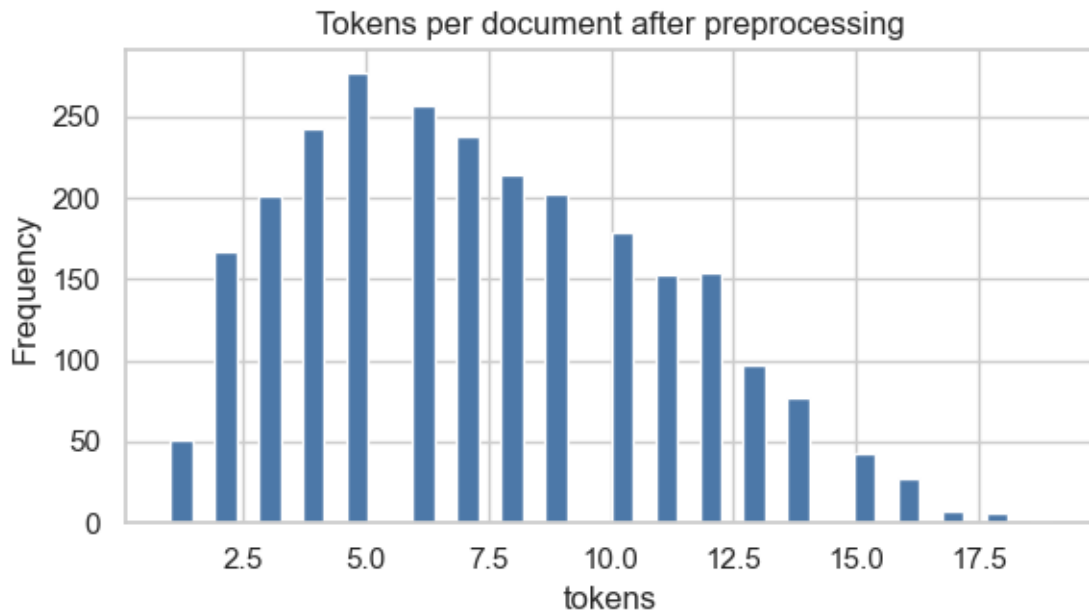
```
[6]: df = preprocess_dataframe(df, TEXT_COL, LABEL_COL, method="stem")
df[[TEXT_COL, "processed_text", LABEL_COL]].head()
```

```
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

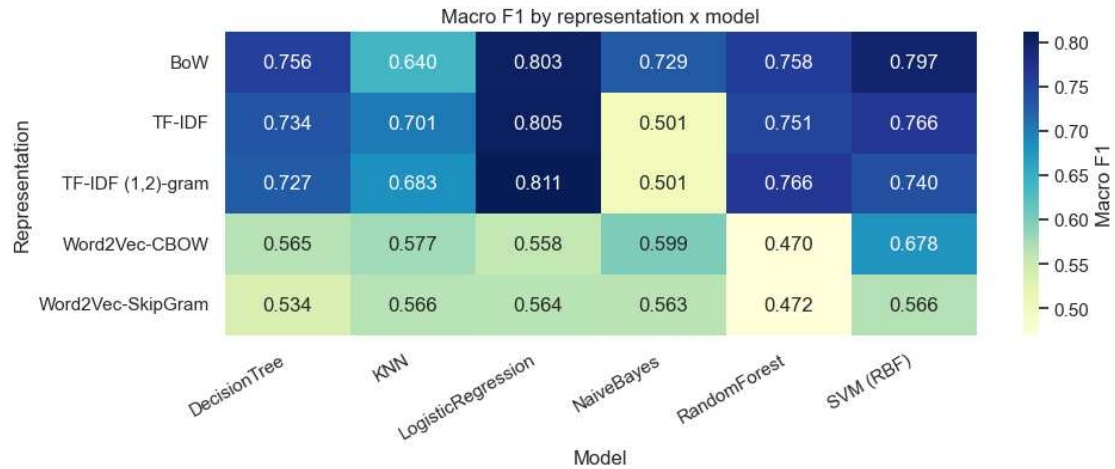
```
[13]: results.head(15)
```

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

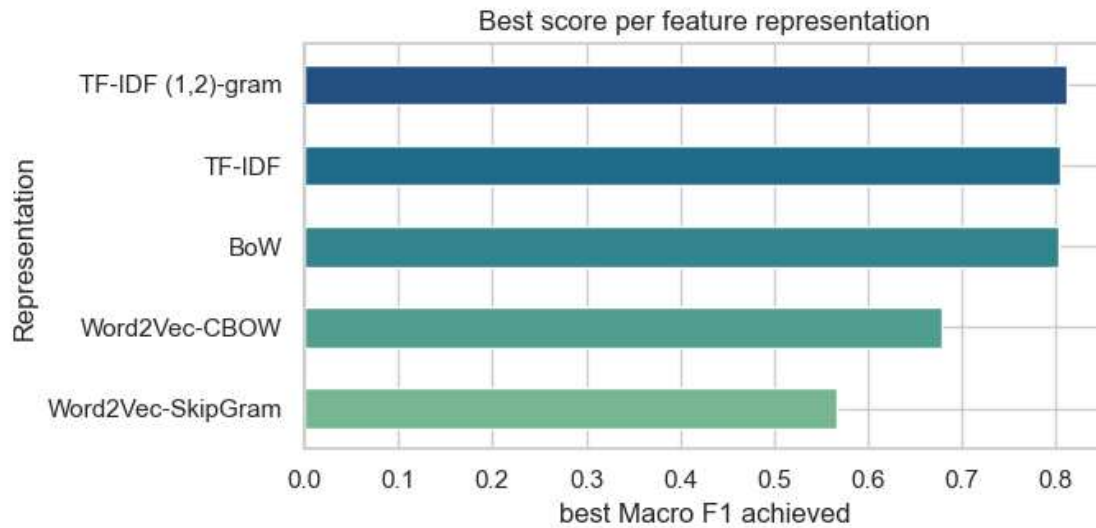
0.861655	0.811221				
1	TF-IDF	LogisticRegression	0.855769	0.863137	0.855769
0.858633	0.805170				
2	BoW	LogisticRegression	0.851923	0.862974	0.851923
0.855884	0.803068				
3	BoW	SVM (RBF)	0.851923	0.856187	0.851923
0.853757	0.796709				
4	TF-IDF	SVM (RBF)	0.857692	0.852857	0.857692
0.844690	0.765543				
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()
```



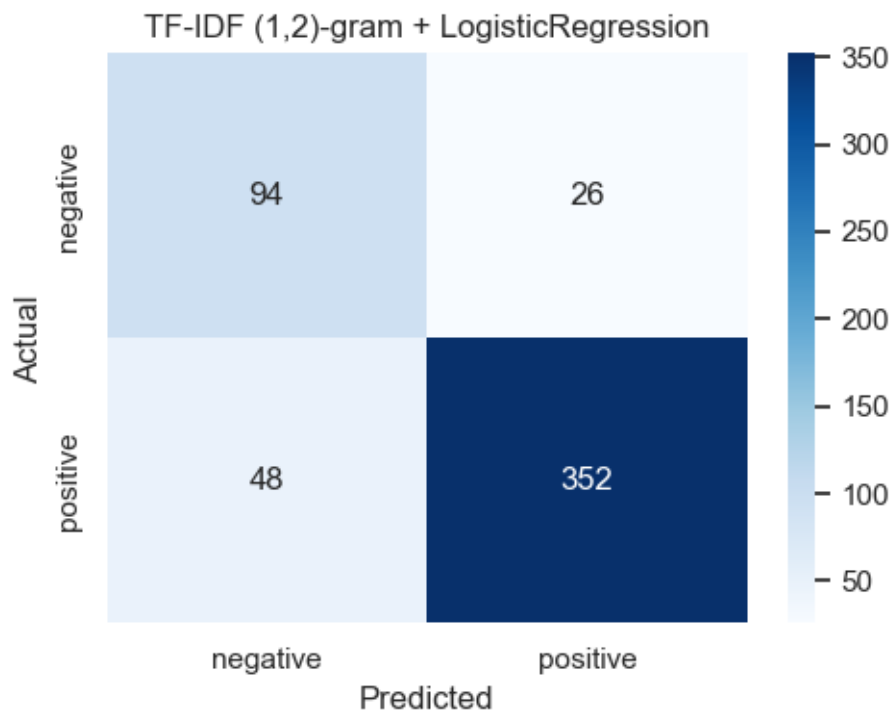
1.6.2 Best model — full report and confusion matrix

```
[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/`.

project2_clean

August 22, 2026

1 Project 2 — Sentiment Classification (Lectures 6–10) — clean_dataset.csv (3 classes)

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

`clean_dataset.csv` was already stemmed by the course, so the cleaning step is close to a no-op here — it mainly re-tokenizes and drops leftover stopwords. Three classes (positive / negative / neutral), ~19.5k rows, so `LinearSVC` replaces `SVC(rbf)`.

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

```
[2]: DATASET, TEXT_COL, LABEL_COL, CLASS_WEIGHT = "clean_dataset.csv", "clean_text",  
      ↪ "sentiment", None  
      # DATASET, TEXT_COL, LABEL_COL, CLASS_WEIGHT = "unbalanceddataset.csv", "text",  
      ↪ "sentiment", "balanced"  
  
df = pd.read_csv(DATASET, encoding="ISO-8859-1")  
print(df.shape)  
df.head()
```

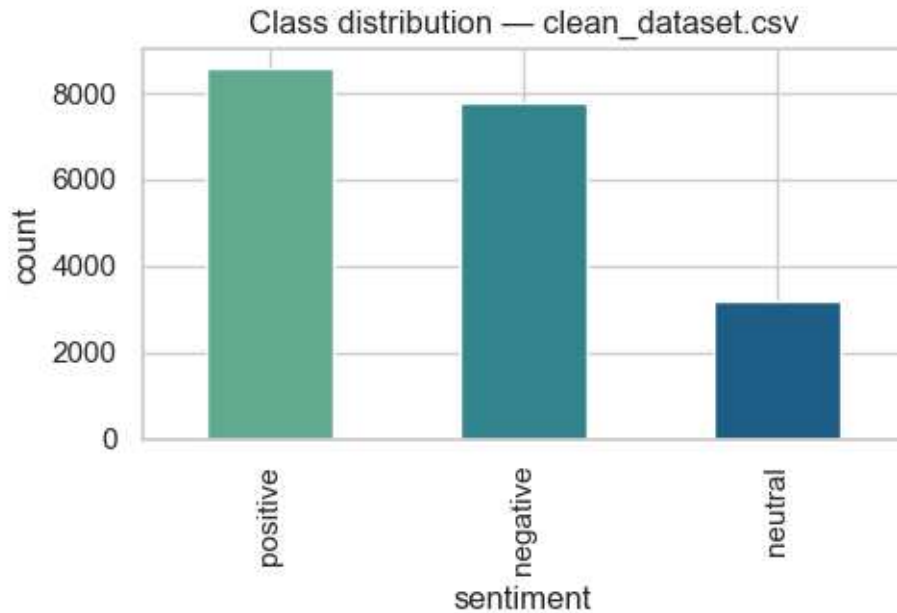
(19557, 2)

```
[2]: sentiment clean_text  
0 negative sooo sad miss san diego  
1 negative boss bulli  
2 negative interview leav alon  
3 negative son couldnt put releas already bought  
4 positive feed babi fun smile coo
```

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

```
[3]: sentiment  
positive    8582  
negative    7781  
neutral     3194  
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: 19557 -> 19127 (dedup/na) -> 19125 (non-empty after cleaning)
avg tokens/doc: 7.3
```

	clean_text	processed_text
sentiment		
0	sooo sad miss san diego	sooo sad miss san diego
negative		
1	boss bulli	boss bulli

```

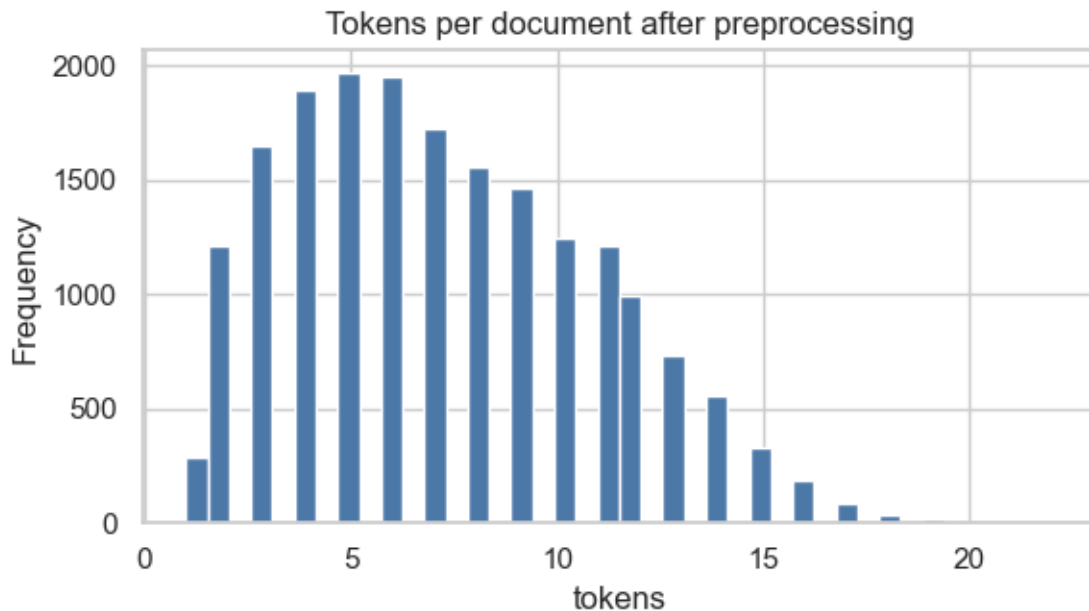
negative
2          interview leav alon          interview leav alon
negative
3 son couldnt put releas already bought son couldnt put relea already bought
negative
4          feed babi fun smile coo          feed babi fun smile coo
positive

```

```

[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    19125.000000
     mean         7.336052
     std         3.705180
     min         1.000000
     25%         4.000000
     50%         7.000000
     75%        10.000000
     max        22.000000
     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=11475  class %: {'positive': 43.5, 'negative': 40.0, 'neutral': 16.4}
val    n=3825  class %: {'positive': 43.5, 'negative': 40.0, 'neutral': 16.4}
test   n=3825  class %: {'positive': 43.5, 'negative': 40.0, 'neutral': 16.4}
```

```
[8]: (11475, 3825, 3825)
```

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 (11475, 5000) (sparse)
TF-IDF       train shape (11475, 5000) (sparse)
TF-IDF (1,2)-gram train shape (11475, 5000) (sparse)
Word2Vec-CBOW train shape (11475, 100) (dense)
Word2Vec-SkipGram train shape (11475, 100) (dense)
Word2Vec vocab: CBOW 4739, SkipGram 4739
```

```
[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: 5000
first 20 features  : ['aaaaaaaaamaz' 'aaaaaalcohol' 'aaaaarrrrrggghhh'
'aaaagggeessss' 'aaaah'
'aaaaiieee' 'aaaau' 'aaaaw' 'aaaawww' 'aaahaha' 'aaahhh' 'aaahhhjust'
'aaarrrrgggghhh' 'aaarrrrgh' 'aaaw' 'aah' 'aargh' 'aaron' 'ab' 'abandon']

```

```

bigrams kept: 1935 -> ['absolut love', 'ahh im', 'alot today', 'alreadi got',
'also want', 'alway end', 'alway forget', 'alway get', 'alway good', 'alway
miss']

```

```

[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: 4739 SkipGram vocab: 4739
most similar to 'good': ['monday', 'nice', 'sunday', 'goodnight', 'first']
most similar to 'bad': ['suck', 'doesnt', 'way', 'didnt', 'either']
most similar to 'love': ['present', 'ever', 'best', 'amaz', 'momma']

```

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=0.7145	weighted-F1=0.7037	macro-F1=0.6293
DecisionTree	acc=0.6787	weighted-F1=0.6827	macro-F1=0.6258

RandomForest	acc=0.7182	weighted-F1=0.7153	macro-F1=0.6541
KNN	acc=0.5516	weighted-F1=0.5759	macro-F1=0.5283
SVM (LinearSVC)	acc=0.6936	weighted-F1=0.6892	macro-F1=0.6219
NaiveBayes	acc=0.7069	weighted-F1=0.6754	macro-F1=0.5774
--- TF-IDF ---			
LogisticRegression	acc=0.7226	weighted-F1=0.6946	macro-F1=0.6048
DecisionTree	acc=0.6675	weighted-F1=0.6687	macro-F1=0.6097
RandomForest	acc=0.7247	weighted-F1=0.7068	macro-F1=0.6286
KNN	acc=0.4118	weighted-F1=0.4338	macro-F1=0.3988
SVM (LinearSVC)	acc=0.7106	weighted-F1=0.6960	macro-F1=0.6185
NaiveBayes	acc=0.7012	weighted-F1=0.6400	macro-F1=0.5128
--- TF-IDF (1,2)-gram ---			
LogisticRegression	acc=0.7299	weighted-F1=0.7017	macro-F1=0.6109
DecisionTree	acc=0.6703	weighted-F1=0.6716	macro-F1=0.6096
RandomForest	acc=0.7231	weighted-F1=0.7048	macro-F1=0.6246
KNN	acc=0.4120	weighted-F1=0.4299	macro-F1=0.3942
SVM (LinearSVC)	acc=0.7148	weighted-F1=0.7023	macro-F1=0.6248
NaiveBayes	acc=0.7090	weighted-F1=0.6485	macro-F1=0.5215
--- Word2Vec-CBOW ---			
LogisticRegression	acc=0.6034	weighted-F1=0.5502	macro-F1=0.4386
DecisionTree	acc=0.4897	weighted-F1=0.4916	macro-F1=0.4319
RandomForest	acc=0.6008	weighted-F1=0.5598	macro-F1=0.4635
KNN	acc=0.5527	weighted-F1=0.5328	macro-F1=0.4542
SVM (LinearSVC)	acc=0.6115	weighted-F1=0.5573	macro-F1=0.4442
NaiveBayes	acc=0.4455	weighted-F1=0.4244	macro-F1=0.3690
--- Word2Vec-SkipGram ---			
LogisticRegression	acc=0.6478	weighted-F1=0.5956	macro-F1=0.4815
DecisionTree	acc=0.5187	weighted-F1=0.5210	macro-F1=0.4623
RandomForest	acc=0.6523	weighted-F1=0.6130	macro-F1=0.5145
KNN	acc=0.5903	weighted-F1=0.5721	macro-F1=0.4894
SVM (LinearSVC)	acc=0.6614	weighted-F1=0.6033	macro-F1=0.4816
NaiveBayes	acc=0.5299	weighted-F1=0.5006	macro-F1=0.4315

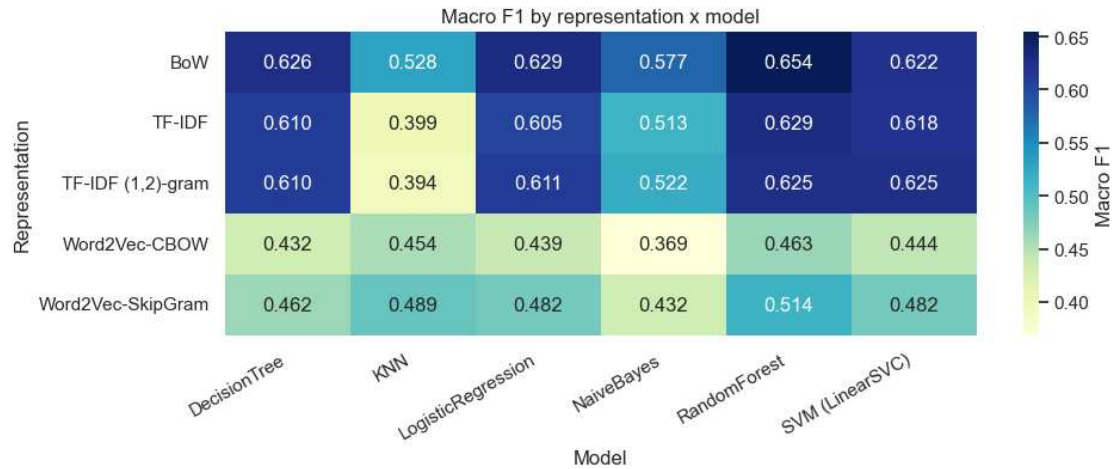
```
[13]: results.head(15)
```

```
[13]:
```

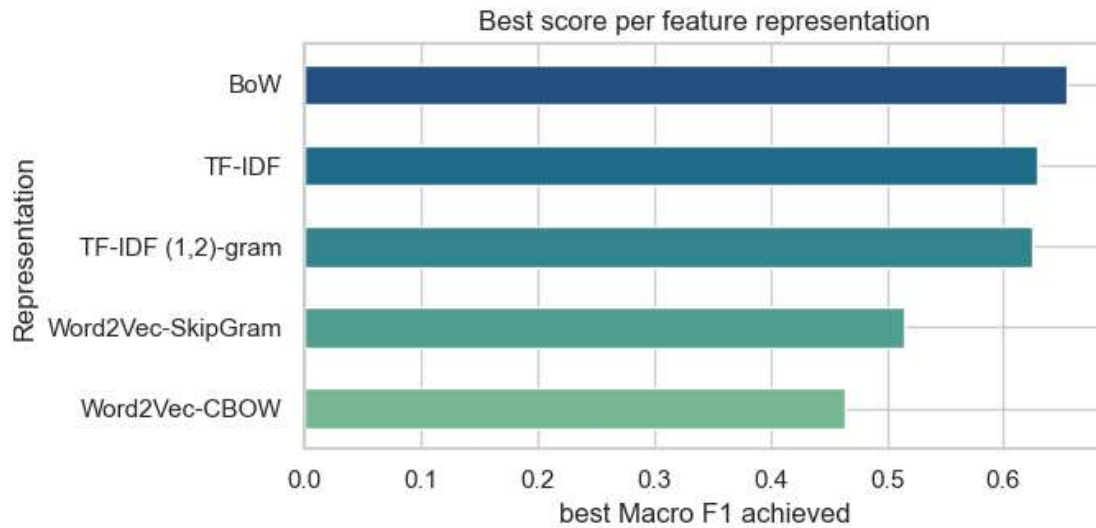
	Representation	Model	Accuracy	Precision	Recall	F1
Score	Macro F1					
0	BoW	RandomForest	0.718170	0.713114	0.718170	
0.715259	0.654083					
1	BoW	LogisticRegression	0.714510	0.698050	0.714510	
0.703680	0.629296					
2	TF-IDF	RandomForest	0.724706	0.704070	0.724706	
0.706792	0.628613					
3	BoW	DecisionTree	0.678693	0.688024	0.678693	
0.682679	0.625775					
4	TF-IDF (1,2)-gram	SVM (LinearSVC)	0.714771	0.696041	0.714771	
0.702338	0.624783					
5	TF-IDF (1,2)-gram	RandomForest	0.723137	0.701778	0.723137	
0.704790	0.624579					
6	BoW	SVM (LinearSVC)	0.693595	0.685740	0.693595	
0.689221	0.621879					
7	TF-IDF	SVM (LinearSVC)	0.710588	0.689984	0.710588	
0.696030	0.618460					
8	TF-IDF (1,2)-gram	LogisticRegression	0.729935	0.705331	0.729935	
0.701682	0.610862					
9	TF-IDF	DecisionTree	0.667451	0.670501	0.667451	
0.668714	0.609739					
10	TF-IDF (1,2)-gram	DecisionTree	0.670327	0.673016	0.670327	
0.671575	0.609645					
11	TF-IDF	LogisticRegression	0.722614	0.697953	0.722614	
0.694636	0.604782					
12	BoW	NaiveBayes	0.706928	0.669423	0.706928	
0.675425	0.577413					
13	BoW	KNN	0.551634	0.630433	0.551634	
0.575884	0.528266					
14	TF-IDF (1,2)-gram	NaiveBayes	0.709020	0.688679	0.709020	
0.648526	0.521527					

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

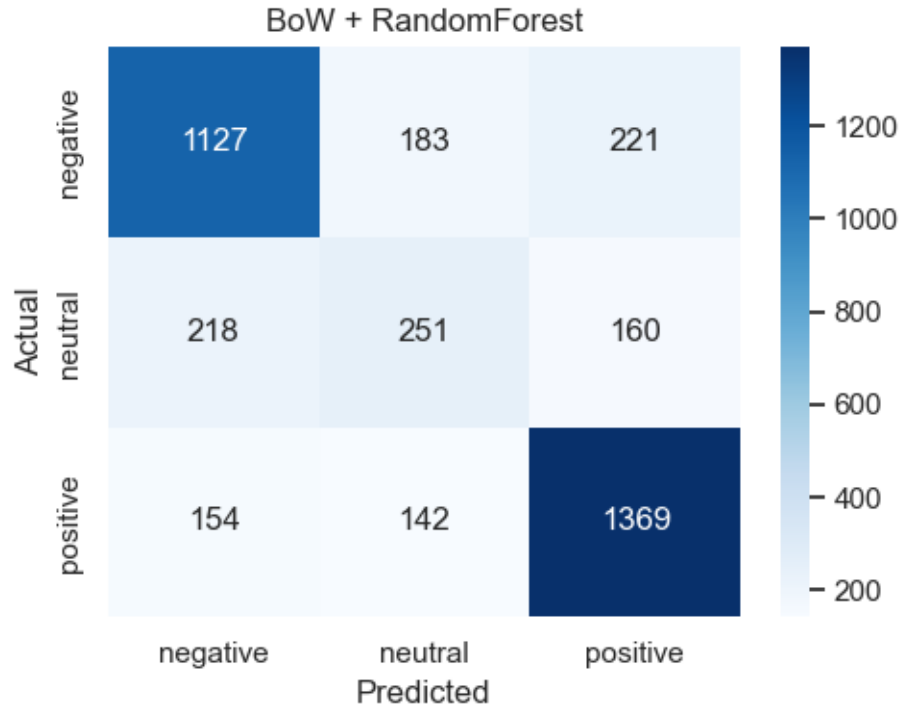
```
[16]: best = results.iloc[0]
res = details[(best["Representation"], best["Model"])]
print(f"BEST: {best['Representation']} + {best['Model']}")
print(f"Accuracy {res['Accuracy']:.4f} | weighted-F1 {res['F1 Score']:.4f} | _
    ↳macro-F1 {res['Macro F1']:.4f}\n")
print(res["Classification Report"])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

BEST: BoW + RandomForest

Accuracy 0.7182 | weighted-F1 0.7153 | macro-F1 0.6541

	precision	recall	f1-score	support
negative	0.75	0.74	0.74	1531
neutral	0.44	0.40	0.42	629
positive	0.78	0.82	0.80	1665
accuracy			0.72	3825
macro avg	0.66	0.65	0.65	3825
weighted avg	0.71	0.72	0.72	3825

```
[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/`.