

Project 2 — Sentiment Classification with Classical ML and Deep Learning

Course: Practical Image Processing and NLP

Task: NLP classification problem — preprocessing, dataset splitting, feature representation, model training and evaluation.

What this notebook does

The project is built on the two sentiment datasets provided with the course:

Dataset	Rows	Text	Classes
unbalanceddataset.csv	2,600	raw, uncleaned tweets/reviews	positive (2000) / negative (600) — strongly imbalanced
clean_dataset.csv	~19,500	already normalised text	positive / negative / neutral — 3 classes

Because the two datasets are complementary, the notebook is organised in two experimental parts:

- **Part A — raw text, binary, imbalanced.** The full preprocessing chain is written from scratch, then *count-based* feature representations (Bag-of-Words and TF-IDF) are compared over four classifiers. The class imbalance is then attacked with over-sampling, under-sampling and SMOTE, and the winning configuration is tuned with a grid search.
- **Part B — larger corpus, three classes.** Here *prediction-based* feature representations are used: Word2Vec embeddings trained with both CBOW and Skip-gram, consumed first as averaged sentence vectors by classical models and then as full sequences by a Bidirectional LSTM.

Every experiment is pushed into one shared results table so that the final comparison is made on exactly the same test protocol.

Reading guide: section 1 setup · 2 data & EDA · 3–6 Part A · 7–10 Part B · 11 overall comparison and conclusions.

1. Environment setup

Run this cell first on Google Colab. `gensim >= 4.4` is required because earlier versions are not compatible with the NumPy 2.x that ships with current Colab runtimes.

In [2]:

```
!pip install -q "gensim>=4.4" imbalanced-learn
```

In [3]:

```
import os
import re
import random
import warnings
from collections import Counter

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

warnings.filterwarnings("ignore")

# -----
# Global configuration – everything that controls the experiments lives here.
# -----
SEED = 42 # single seed reused everywhere so results are reproducible
TEST_SIZE = 0.30 # hold-out fraction for Part A
TEST_SIZE_B = 0.20 # hold-out fraction for Part B
EMBED_DIM = 100 # Word2Vec vector size
W2V_WINDOW = 5
MAX_LEN = 25 # sequence length fed to the LSTM
QUICK_RUN = False # True -> small subsample + tiny grids, used only to smoke-test the code

random.seed(SEED)
np.random.seed(SEED)

plt.rcParams["figure.figsize"] = (7, 4)
plt.rcParams["axes.grid"] = True
plt.rcParams["grid.alpha"] = 0.3
```

In [4]:

```
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer

for pkg in ["punkt", "punkt_tab", "stopwords", "wordnet"]:
    try:
        nltk.download(pkg, quiet=True)
    except Exception as exc:
        print(f"could not download {pkg}: {exc}")

try:
    from nltk.tokenize import word_tokenize
    word_tokenize("smoke test") # verifies the tokeniser data is really there
    TOKENIZER = word_tokenize
except Exception:
    print("NLTK tokeniser unavailable – using a regular-expression tokeniser instead.")
    TOKENIZER = lambda s: re.findall(r"[a-z]+", s.lower())

try:
    STOPWORDS = set(stopwords.words("english"))
except Exception:
    STOPWORDS = set(
        "i me my myself we our ours ourselves you your yours yourself yourselves he him his "
        "himself she her hers herself it its itself they them their theirs themselves what "
```

```

        "which who whom this that these those am is are was were be been being have has had "
        "having do does did doing a an the and but if or because as until while of at by for "
        "with about against between into through during before after above below to from up "
        "down in out on off over under again further then once here there when where why how "
        "all any both each few more most other some such no nor not only own same so than too "
        "very s t can will just don should now".split()
    )

    STEMMER = PorterStemmer()
    try:
        LEMMATIZER = WordNetLemmatizer()
        LEMMATIZER.lemmatize("tests")
    except Exception:
        LEMMATIZER = None

    print(f"stop-words loaded: {len(STOPWORDS)} | lemmatizer: {'yes' if LEMMATIZER else 'no'}")

```

stop-words loaded: 198 | lemmatizer: yes

In [5]:

```

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import LinearSVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder
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 gensim.models import Word2Vec

```

1.1 Shared evaluation helpers

One evaluation function is used by every experiment in the notebook. Besides accuracy it reports the **macro F1**, which is the honest metric when the classes are imbalanced: the weighted average can look excellent while the minority class is being ignored completely.

In [6]:

```
RESULTS = [] # every experiment appends one row here

def evaluate(experiment, part, y_true, y_pred, class_names=None, show_report=True, plot=True):
    """Score one experiment, print a report, draw its confusion matrix and store the row."""
    row = {
        "Experiment": experiment,
        "Part": part,
        "Accuracy": accuracy_score(y_true, y_pred),
        "Precision": precision_score(y_true, y_pred, average="weighted", zero_division=0),
        "Recall": recall_score(y_true, y_pred, average="weighted", zero_division=0),
        "F1 (weighted)": f1_score(y_true, y_pred, average="weighted", zero_division=0),
        "F1 (macro)": f1_score(y_true, y_pred, average="macro", zero_division=0),
    }
    RESULTS.append(row)

    print(f"\n=== {experiment} ===")
    print(f"Accuracy {row['Accuracy']:.4f} | weighted F1 {row['F1 (weighted)']:.4f} "
          f"| macro F1 {row['F1 (macro)']:.4f}")
    if show_report:
        print(classification_report(y_true, y_pred, target_names=class_names, zero_division=0))
    if plot:
        plot_confusion(y_true, y_pred, class_names, experiment)
    return row

def plot_confusion(y_true, y_pred, class_names, title):
    """Confusion matrix drawn with matplotlib only (no extra dependency)."""
    cm = confusion_matrix(y_true, y_pred)
    labels = class_names if class_names is not None else sorted(set(np.concatenate([y_true, y_pred])))
    fig, ax = plt.subplots(figsize=(4.2, 3.6))
    im = ax.imshow(cm, cmap="Blues")
    ax.set_xticks(range(len(labels)), labels, rotation=45, ha="right")
    ax.set_yticks(range(len(labels)), labels)
    ax.set_xlabel("Predicted"); ax.set_ylabel("True"); ax.set_title(title, fontsize=10)
    ax.grid(False)
    thresh = cm.max() / 2
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, cm[i, j], ha="center", va="center",
                    color="white" if cm[i, j] > thresh else "black", fontsize=9)
    fig.colorbar(im, ax=ax, shrink=0.8)
    plt.tight_layout(); plt.show()

def results_table(part=None):
    """Current leaderboard, sorted by macro F1."""
    df = pd.DataFrame(RESULTS)
    if part is not None:
        df = df[df["Part"] == part]
    return (df.drop(columns="Part")
            .sort_values("F1 (macro)", ascending=False)
            .reset_index(drop=True)
            .round(4))
```

2. Loading the datasets

On Colab the two CSV files are uploaded once with the file picker; the helper below also looks in the working directory and in Google Drive so the same notebook runs locally without any change.

In [7]:

```
def resolve_path(*candidates):
    """Return the first existing path among the candidates, otherwise ask Colab to upload."""
    search_dirs = [".", "/content", "/content/drive/MyDrive", "/content/drive/MyDrive/data"]
    for name in candidates:
        for folder in search_dirs:
            path = os.path.join(folder, name)
            if os.path.exists(path):
                return path

    try:
        from google.colab import files
        print(f"Please upload: {candidates[0]}")
        uploaded = files.upload()
        return list(uploaded.keys())[0]
    except ImportError:
        raise FileNotFoundError(f"None of {candidates} was found in {search_dirs}")

RAW_PATH = resolve_path("unbalanceddataset.csv", "unbalanceddataset (2).csv")
CLEAN_PATH = resolve_path("clean_dataset.csv", "clean_dataset (3).csv")

raw_df = pd.read_csv(RAW_PATH, encoding="ISO-8859-1")
clean_df = pd.read_csv(CLEAN_PATH, encoding="ISO-8859-1")

print("raw dataset :", raw_df.shape, list(raw_df.columns))
print("clean dataset:", clean_df.shape, list(clean_df.columns))
raw_df.head()
```

```
raw dataset : (2600, 2) ['text', 'sentiment']
clean dataset: (19557, 2) ['sentiment', 'clean_text']
```

Out[7]:

```
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
```

2.1 Data quality check and cleaning of the frames themselves

Before any text processing, the two frames are checked for missing values and duplicated rows. Duplicates matter more than usual in text classification: if the same sentence exists twice and the split puts one copy in train and one in test, the test score is inflated.

In [8]:

```
def frame_report(df, name, text_col, label_col):
    print(f"--- {name} ---")
    print("shape      :", df.shape)
    print("missing values:", df.isna().sum().to_dict())
    print("duplicated rows:", int(df.duplicated().sum()))
    print("class counts  :", df[label_col].value_counts().to_dict())
    print("empty strings  :", int((df[text_col].astype(str).str.strip() == "").sum()))
    print()

frame_report(raw_df, "unbalanced (raw)", "text", "sentiment")
frame_report(clean_df, "clean (3 classes)", "clean_text", "sentiment")

raw_df = raw_df.dropna(subset=["text", "sentiment"]).drop_duplicates().reset_index(drop=True)
clean_df = (clean_df.dropna(subset=["clean_text", "sentiment"])
            .drop_duplicates()
            .reset_index(drop=True))
print("after cleaning ->", raw_df.shape, clean_df.shape)

if QUICK_RUN:
    # only used when smoke-testing the code
    raw_df = raw_df.sample(400, random_state=SEED).reset_index(drop=True)
    clean_df = clean_df.sample(2000, random_state=SEED).reset_index(drop=True)
    print("QUICK_RUN subsample ->", raw_df.shape, clean_df.shape)
```

```
--- unbalanced (raw) ---
shape      : (2600, 2)
missing values : {'text': 0, 'sentiment': 0}
duplicated rows: 0
class counts  : {'positive': 2000, 'negative': 600}
empty strings  : 0

--- clean (3 classes) ---
shape      : (19557, 2)
missing values : {'sentiment': 0, 'clean_text': 22}
duplicated rows: 404
class counts  : {'positive': 8582, 'negative': 7781, 'neutral': 3194}
empty strings  : 0

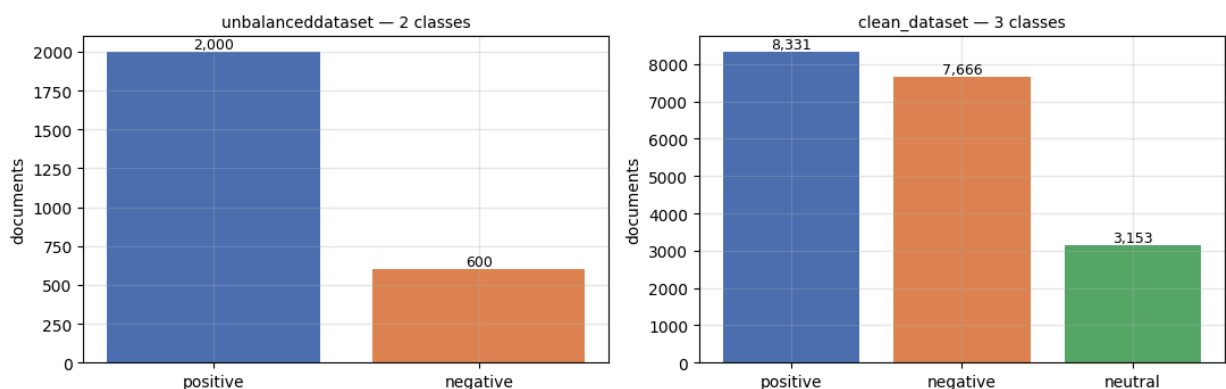
after cleaning -> (2600, 2) (19150, 2)
```

2.2 Exploratory data analysis

In [9]:

```
fig, axes = plt.subplots(1, 2, figsize=(11, 3.6))
for ax, (df, col, title) in zip(
    axes,
    [(raw_df, "sentiment", "unbalanceddataset — 2 classes"),
     (clean_df, "sentiment", "clean_dataset — 3 classes")],
):
    counts = df[col].value_counts()
    ax.bar(counts.index, counts.values, color=["#4C72B0", "#DD8452", "#55A868"][0: len(counts)])
    ax.set_title(title, fontsize=10); ax.set_ylabel("documents")
    for i, v in enumerate(counts.values):
        ax.text(i, v, f"{v:,}", ha="center", va="bottom", fontsize=9)
plt.tight_layout(); plt.show()

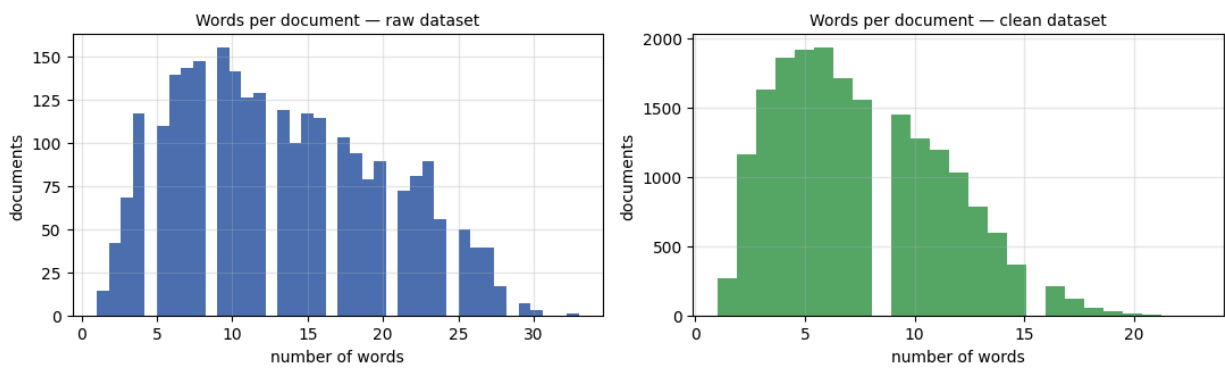
imbalance_ratio = raw_df["sentiment"].value_counts()
print(f"Imbalance ratio of the raw dataset: "
      f"{imbalance_ratio.max() / imbalance_ratio.min():.2f} : 1")
```



Imbalance ratio of the raw dataset: 3.33 : 1

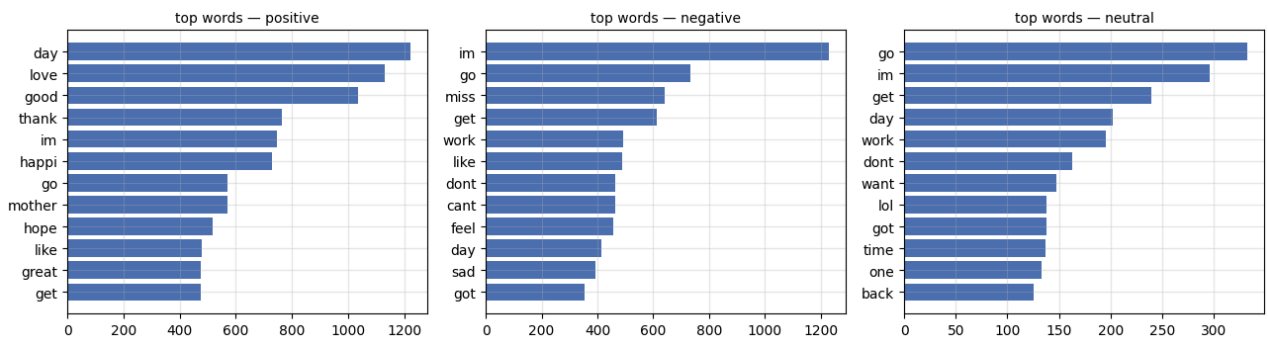
In [10]:

```
fig, axes = plt.subplots(1, 2, figsize=(11, 3.4))
axes[0].hist(raw_df["text"].str.split().str.len(), bins=40, color="#4C72B0")
axes[0].set_title("Words per document — raw dataset", fontsize=10)
axes[1].hist(clean_df["clean_text"].astype(str).str.split().str.len(), bins=25, color="#55A868")
axes[1].set_title("Words per document — clean dataset", fontsize=10)
for ax in axes:
    ax.set_xlabel("number of words"); ax.set_ylabel("documents")
plt.tight_layout(); plt.show()
```



```
In [11]:
def top_words(series, k=15):
    counter = Counter()
    for doc in series.astype(str):
        counter.update(w for w in TOKENIZER(doc.lower()) if w.isalpha() and w not in STOPWORDS)
    return counter.most_common(k)

fig, axes = plt.subplots(1, 3, figsize=(13, 3.6))
for ax, label in zip(axes, ["positive", "negative", "neutral"]):
    subset = clean_df.loc[clean_df["sentiment"] == label, "clean_text"]
    if subset.empty:
        ax.axis("off"); continue
    words, counts = zip(*top_words(subset, 12))
    ax.barh(list(words)[:1], list(counts)[:1], color="#4C72B0")
    ax.set_title(f"top words - {label}", fontsize=10)
plt.tight_layout(); plt.show()
```



What the EDA tells us

- The raw dataset is imbalanced roughly 3.3 : 1 in favour of `positive` ; a model that always answers *positive* would already reach ~77 % accuracy, so accuracy alone will be misleading.
- The clean dataset is more balanced between `positive` and `negative` , but `neutral` is a clear minority and — as the lecture results showed — it is the class every model struggles with, because neutral text shares its vocabulary with both other classes.
- Documents are short (median $\approx$ 7-12 words). Short texts favour sparse count features and put an upper bound on how much a sequence model can learn.

Part A — Raw text · binary classification · imbalanced data

3. Text preprocessing

The cleaning chain applied to every document:

1. lower-casing;
2. removal of URLs and HTML tags;
3. removal of user mentions (`@user`) and the `#` of hashtags — the word itself is kept;
4. removal of everything that is not a letter (digits, punctuation, emoticons);
5. tokenisation;
6. removal of stop-words and of tokens shorter than three characters;
7. normalisation — **lemmatisation** (dictionary form) or **stemming** (suffix chopping).

Both normalisation strategies are implemented so that their effect can be measured instead of assumed. Negations (`not` , `no` , `never`) are deliberately *kept*, even though they belong to the standard stop-word list — dropping them turns "not good" into "good".

```
In [12]:
NEGATIONS = {"not", "no", "nor", "never", "cannot", "cant", "dont", "doesnt", "didnt",
             "isnt", "wasnt", "arent", "werent", "wont", "wouldnt", "couldnt", "shouldnt"}
ACTIVE_STOPWORDS = STOPWORDS - NEGATIONS

URL_RE = re.compile(r"http\S+|www\.\S+")
HTML_RE = re.compile(r"<.*>")
MENTION_RE = re.compile(r"@w+")
```

```

NONALPHA_RE = re.compile(r"[^a-z\s]")

def preprocess(text, normalise="lemma", min_len=3):
    """Full cleaning chain: raw string -> normalised, space-joined tokens."""
    text = str(text).lower()
    text = URL_RE.sub(" ", text)
    text = HTML_RE.sub(" ", text)
    text = MENTION_RE.sub(" ", text)
    text = text.replace("#", " ")
    text = NONALPHA_RE.sub(" ", text)

    tokens = [t for t in TOKENIZER(text) if t not in ACTIVE_STOPWORDS and len(t) >= min_len]

    if normalise == "stem":
        tokens = [STEMMER.stem(t) for t in tokens]
    elif normalise == "lemma" and LEMMATIZER is not None:
        tokens = [LEMMATIZER.lemmatize(t, pos="v") for t in tokens]
    return " ".join(tokens)

# side-by-side look at what the chain actually does
demo = raw_df["text"].head(4).tolist()
pd.set_option("display.max_colwidth", 70)
pd.DataFrame({
    "original": demo,
    "lemmatised": [preprocess(t, "lemma") for t in demo],
    "stemmed": [preprocess(t, "stem") for t in demo],
})

```

```

Out[12]:

```

	original	lemmatised	stemmed
0	Java Concurrency in Practice is probably the best Java book I've e...	java concurrency practice probably best java book ever buy recipe ...	java concurr practic probabl best java book ever bought recip inte...
1	haha aww hun i bet you are more creative than me	haha aww hun bet creative	haha aww hun bet creativ
2	_pickle lol, thank you very much Hope you're having a great day!	pickle lol thank much hope great day	pickl lol thank much hope great day
3	Out for an evening on the town with jeremy. Sad Carrie can't come	even town jeremy sad carrie come	even town jeremi sad carri come

```

In [13]:
raw_df["text_lemma"] = raw_df["text"].apply(lambda t: preprocess(t, "lemma"))
raw_df["text_stem"] = raw_df["text"].apply(lambda t: preprocess(t, "stem"))

# documents that became empty after cleaning carry no signal at all
empty = (raw_df["text_lemma"].str.strip() == "")
print("documents emptied by cleaning:", int(empty.sum()))
raw_df = raw_df[~empty].reset_index(drop=True)
print("final size of Part A dataset :", raw_df.shape)
raw_df[["text", "text_lemma", "sentiment"]].head()

```

documents emptied by cleaning: 3
final size of Part A dataset : (2597, 4)

```

Out[13]:

```

	text	text_lemma	sentiment
0	Java Concurrency in Practice is probably the best Java book I've e...	java concurrency practice probably best java book ever buy recipe ...	positive
1	haha aww hun i bet you are more creative than me	haha aww hun bet creative	positive
2	_pickle lol, thank you very much Hope you're having a great day!	pickle lol thank much hope great day	positive
3	Out for an evening on the town with jeremy. Sad Carrie can't come	even town jeremy sad carrie come	negative
4	- just took over the #1 Most Endorsed spot on twindexx.com - than...	take endorse spot twindexx com thank endorsement	positive

4. Splitting the dataset

A single **stratified** 70 / 30 split is created and reused by every Part A experiment, so that all the numbers in the results table are comparable.

Stratification keeps the 77 / 23 class ratio identical in train and test — without it the small **negative** class could be badly under-represented in the test set purely by chance.

Rule respected throughout the notebook: vectorisers, embeddings and resamplers are fitted on the **training set only**. Fitting them on the whole dataset would leak test-set vocabulary into training and inflate the scores.

```

In [14]:
X_raw_train, X_raw_test, y_a_train, y_a_test = train_test_split(
    raw_df[["text_lemma", "text_stem"]], raw_df["sentiment"],
    test_size=TEST_SIZE, random_state=SEED, stratify=raw_df["sentiment"],
)
print("train:", X_raw_train.shape[0], dict(Counter(y_a_train)))
print("test :", X_raw_test.shape[0], dict(Counter(y_a_test)))

```

train: 1817 {'positive': 1398, 'negative': 419}
test : 780 {'positive': 600, 'negative': 180}

5. Feature representation — count-based methods

Two classical representations are compared:

- **Bag-of-Words (CountVectorizer)** — raw term counts. Simple, but frequent words dominate.
- **TF-IDF (TfidfVectorizer)** — counts weighted down by how common the term is across the corpus, so words that discriminate between classes get a larger weight.

Both are built with uni-grams *and* bi-grams (ngram_range=(1, 2)) so that expressions such as "not good" survive as a single feature, and with min_df=2 to drop terms that appear in a single document (mostly typos, which only add noise and dimensions).

```

In [15]:
def build_features(vectorizer, train_texts, test_texts):
    """Fit the vectoriser on train only, transform both sides."""
    Xtr = vectorizer.fit_transform(train_texts)
    Xte = vectorizer.transform(test_texts)

```

```

return Xtr, Xte, vectorizer

bow_vec = CountVectorizer(ngram_range=(1, 2), min_df=2)
tfidf_vec = TfidfVectorizer(ngram_range=(1, 2), min_df=2, sublinear_tf=True)

X_bow_tr, X_bow_te, bow_vec = build_features(bow_vec, X_raw_train["text_lemma"], X_raw_test["text_lemma"])
X_tfidf_tr, X_tfidf_te, tfidf_vec = build_features(tfidf_vec, X_raw_train["text_lemma"], X_raw_test["text_lemma"])

# the same TF-IDF settings on the *stemmed* text, to measure lemmatisation vs stemming
stem_vec = TfidfVectorizer(ngram_range=(1, 2), min_df=2, sublinear_tf=True)
X_stem_tr, X_stem_te, stem_vec = build_features(stem_vec, X_raw_train["text_stem"], X_raw_test["text_stem"])

print(f"BoW vocabulary: {len(bow_vec.vocabulary_):>6,} matrix {X_bow_tr.shape}")
print(f"TFIDF vocabulary: {len(tfidf_vec.vocabulary_):>6,} matrix {X_tfidf_tr.shape}")
print(f"TFIDF (stemmed) : {len(stem_vec.vocabulary_):>6,} matrix {X_stem_tr.shape}")

BoW vocabulary: 1,761 matrix (1817, 1761)
TFIDF vocabulary: 1,761 matrix (1817, 1761)
TFIDF (stemmed) : 1,786 matrix (1817, 1786)

```

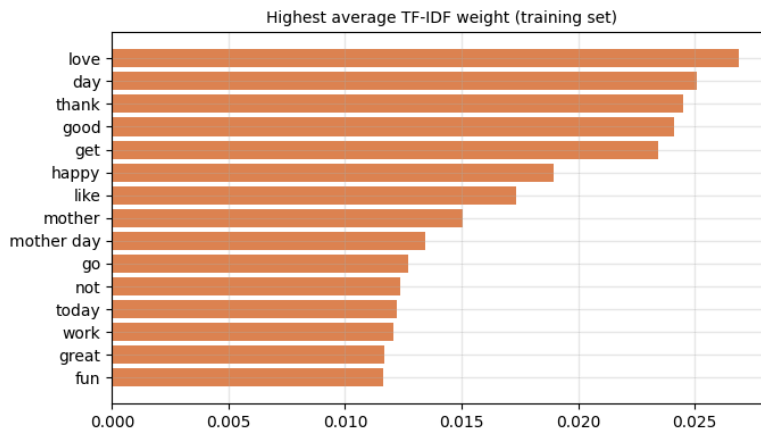
5.1 Which terms does TF-IDF consider most informative?

In [16]:

```

tfidf_scores = np.asarray(X_tfidf_tr.mean(axis=0)).ravel()
vocab = np.array(tfidf_vec.get_feature_names_out())
top_idx = tfidf_scores.argsort()[-15:]
plt.barh(vocab[top_idx], tfidf_scores[top_idx], color="#DD8452")
plt.title("Highest average TF-IDF weight (training set)", fontsize=10)
plt.tight_layout(); plt.show()

```



6. Training and evaluating the models

Four classifiers with genuinely different inductive biases are compared on both representations:

Model	Why it is in the comparison
Logistic Regression	strong, fast linear baseline for sparse text features
Multinomial Naive Bayes	the classical text-classification model; assumes term independence
Linear SVM	usually the best linear model on high-dimensional sparse data
Random Forest	non-linear, captures term interactions, but suffers in very high dimensions

In [17]:

```

def make_models():
    """Fresh, unfitted model instances (a fitted model must never be reused across runs)."""
    return {
        "LogisticRegression": LogisticRegression(max_iter=2000, random_state=SEED),
        "MultinomialNB": MultinomialNB(),
        "LinearSVC": LinearSVC(random_state=SEED),
        "RandomForest": RandomForestClassifier(
            n_estimators=100 if QUICK_RUN else 300, random_state=SEED, n_jobs=-1),
    }

def run_models(Xtr, ytr, Xte, yte, tag, part, class_names, plot_for=()):
    """Train every model on one representation and record the scores."""
    for name, model in make_models().items():
        model.fit(Xtr, ytr)
        evaluate(f"{name} + {tag}", part, yte, model.predict(Xte),
                class_names=class_names, show_report=name in plot_for, plot=name in plot_for)

CLASSES_A = sorted(raw_df["sentiment"].unique())

run_models(X_bow_tr, y_a_train, X_bow_te, y_a_test, "BoW (lemma)", "A", CLASSES_A)
run_models(X_tfidf_tr, y_a_train, X_tfidf_te, y_a_test, "TF-IDF (lemma)", "A", CLASSES_A,
            plot_for=("LinearSVC",))
run_models(X_stem_tr, y_a_train, X_stem_te, y_a_test, "TF-IDF (stem)", "A", CLASSES_A)

results_table("A")

```

```

=== LogisticRegression + BoW (lemma) ===
Accuracy 0.8526 | weighted F1 0.8406 | macro F1 0.7609

=== MultinomialNB + BoW (lemma) ===
Accuracy 0.8603 | weighted F1 0.8524 | macro F1 0.7822

=== LinearSVC + BoW (lemma) ===
Accuracy 0.8526 | weighted F1 0.8512 | macro F1 0.7886

=== RandomForest + BoW (lemma) ===
Accuracy 0.8487 | weighted F1 0.8444 | macro F1 0.7752

=== LogisticRegression + TF-IDF (lemma) ===
Accuracy 0.8308 | weighted F1 0.7942 | macro F1 0.6705

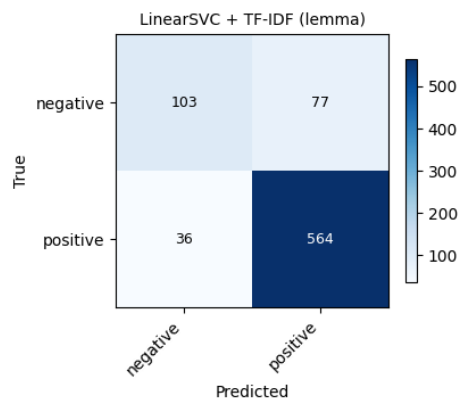
=== MultinomialNB + TF-IDF (lemma) ===
Accuracy 0.8115 | weighted F1 0.7568 | macro F1 0.6004

=== LinearSVC + TF-IDF (lemma) ===
Accuracy 0.8551 | weighted F1 0.8482 | macro F1 0.7774
precision    recall  f1-score   support

   negative     0.74     0.57     0.65       180
   positive     0.88     0.94     0.91       600

   accuracy          0.81     0.76     0.86       780
  macro avg          0.81     0.76     0.78       780
weighted avg          0.85     0.86     0.85       780

```



```

=== RandomForest + TF-IDF (lemma) ===
Accuracy 0.8564 | weighted F1 0.8435 | macro F1 0.7641

=== LogisticRegression + TF-IDF (stem) ===
Accuracy 0.8321 | weighted F1 0.7953 | macro F1 0.6718

=== MultinomialNB + TF-IDF (stem) ===
Accuracy 0.8154 | weighted F1 0.7637 | macro F1 0.6131

=== LinearSVC + TF-IDF (stem) ===
Accuracy 0.8615 | weighted F1 0.8547 | macro F1 0.7867

=== RandomForest + TF-IDF (stem) ===
Accuracy 0.8526 | weighted F1 0.8406 | macro F1 0.7609

```

```

Out[17]:
Experiment Accuracy Precision Recall F1 (weighted) F1 (macro)
0 LinearSVC + BoW (lemma) 0.8526 0.8502 0.8526 0.8512 0.7886
1 LinearSVC + TF-IDF (stem) 0.8615 0.8550 0.8615 0.8547 0.7867
2 MultinomialNB + BoW (lemma) 0.8603 0.8535 0.8603 0.8524 0.7822
3 LinearSVC + TF-IDF (lemma) 0.8551 0.8478 0.8551 0.8482 0.7774
4 RandomForest + BoW (lemma) 0.8487 0.8426 0.8487 0.8444 0.7752
5 RandomForest + TF-IDF (lemma) 0.8564 0.8510 0.8564 0.8435 0.7641
6 LogisticRegression + BoW (lemma) 0.8526 0.8455 0.8526 0.8406 0.7609
7 RandomForest + TF-IDF (stem) 0.8526 0.8455 0.8526 0.8406 0.7609
8 LogisticRegression + TF-IDF (stem) 0.8321 0.8516 0.8321 0.7953 0.6718
9 LogisticRegression + TF-IDF (lemma) 0.8308 0.8475 0.8308 0.7942 0.6705
10 MultinomialNB + TF-IDF (stem) 0.8154 0.8511 0.8154 0.7637 0.6131
11 MultinomialNB + TF-IDF (lemma) 0.8115 0.8486 0.8115 0.7568 0.6004

```

6.1 The imbalance problem

The table above shows the trap of imbalanced data: accuracy looks respectable while the macro F1 stays far behind it, because the models mostly predict the majority class. Three standard remedies are now applied **to the training set only** — resampling the test set would make the evaluation meaningless:

- **Random over-sampling** — duplicate minority examples; no information is lost, but the duplicated rows encourage over-fitting.
- **Random under-sampling** — drop majority examples; fast, but throws away real data.
- **SMOTE** — synthesise new minority points by interpolating between a minority sample and one of its k nearest minority neighbours, so the new points are not exact copies.
- **class_weight="balanced"** — no resampling at all: the loss function simply penalises minority errors more. Often the cheapest and strongest option.

```

In [18]:
samplers = {
    "no_resampling": None,
    "RandomOverSampler": RandomOverSampler(random_state=SEED),
    "RandomUnderSampler": RandomUnderSampler(random_state=SEED),
    "SMOTE": SMOTE(random_state=SEED, k_neighbors=5),
}

for label, sampler in samplers.items():
    if sampler is None:
        Xs, ys = X_tfidf_tr, y_a_train

```

```

else:
    Xs, ys = sampler.fit_resample(X_tfidf_tr, y_a_train)
    print(f"{label:<20} training distribution: {dict(Counter(ys))}")

    model = LinearSVC(random_state=SEED).fit(Xs, ys)
    evaluate(f"LinearSVC + TF-IDF + {label}", "A", y_a_test, model.predict(X_tfidf_te),
            class_names=CLASSES_A, show_report=False, plot=False)

weighted = LinearSVC(class_weight="balanced", random_state=SEED).fit(X_tfidf_tr, y_a_train)
evaluate("LinearSVC + TF-IDF + class_weight", "A", y_a_test, weighted.predict(X_tfidf_te),
        class_names=CLASSES_A, show_report=True, plot=True)

results_table("A").head(8)

```

no resampling training distribution: {'positive': 1398, 'negative': 419}

```

=== LinearSVC + TF-IDF + no resampling ===
Accuracy 0.8551 | weighted F1 0.8482 | macro F1 0.7774
RandomOverSampler    training distribution: {'positive': 1398, 'negative': 1398}

```

```

=== LinearSVC + TF-IDF + RandomOverSampler ===
Accuracy 0.8436 | weighted F1 0.8450 | macro F1 0.7839
RandomUnderSampler   training distribution: {'negative': 419, 'positive': 419}

```

```

=== LinearSVC + TF-IDF + RandomUnderSampler ===
Accuracy 0.7846 | weighted F1 0.7976 | macro F1 0.7383
SMOTE                training distribution: {'positive': 1398, 'negative': 1398}

```

```

=== LinearSVC + TF-IDF + SMOTE ===
Accuracy 0.8423 | weighted F1 0.8471 | macro F1 0.7922

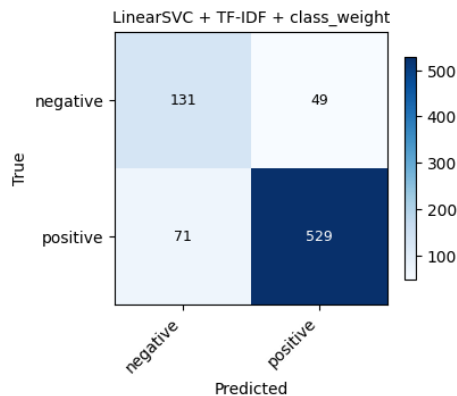
```

```

=== LinearSVC + TF-IDF + class_weight ===
Accuracy 0.8462 | weighted F1 0.8491 | macro F1 0.7920
precision    recall    f1-score    support

```

negative	0.65	0.73	0.69	180
positive	0.92	0.88	0.90	600
accuracy			0.85	780
macro avg	0.78	0.80	0.79	780
weighted avg	0.85	0.85	0.85	780



```

Out[18]:
Experiment  Accuracy  Precision  Recall  F1 (weighted)  F1 (macro)
0  LinearSVC + TF-IDF + SMOTE  0.8423  0.8562  0.8423  0.8471  0.7922
1  LinearSVC + TF-IDF + class_weight  0.8462  0.8537  0.8462  0.8491  0.7920
2  LinearSVC + BoW (lemma)  0.8526  0.8502  0.8526  0.8512  0.7886
3  LinearSVC + TF-IDF (stem)  0.8615  0.8550  0.8615  0.8547  0.7867
4  LinearSVC + TF-IDF + RandomOverSampler  0.8436  0.8468  0.8436  0.8450  0.7839
5  MultinomialNB + BoW (lemma)  0.8603  0.8535  0.8603  0.8524  0.7822
6  LinearSVC + TF-IDF (lemma)  0.8551  0.8478  0.8551  0.8482  0.7774
7  LinearSVC + TF-IDF + no resampling  0.8551  0.8478  0.8551  0.8482  0.7774

```

6.2 Hyper-parameter tuning of the best configuration

`GridSearchCV` searches the parameter grid with 5-fold **stratified** cross-validation inside the training set, scoring on macro F1 rather than accuracy so the minority class actually counts. Only the winning combination is then measured once on the untouched test set.

```

In [19]:
param_grid = (
    {"C": [0.1, 1]} if QUICK_RUN else
    {"C": [0.01, 0.1, 0.5, 1, 5], "class_weight": [None, "balanced"], "loss": ["squared_hinge"]}
)

grid = GridSearchCV(LinearSVC(random_state=SEED), param_grid,
                    cv=3 if QUICK_RUN else 5, scoring="f1_macro", n_jobs=-1)
grid.fit(X_tfidf_tr, y_a_train)

print("best parameters   :", grid.best_params_)
print("best CV macro F1  :", round(grid.best_score_, 4))

_ = evaluate("LinearSVC + TF-IDF + GridSearchCV", "A", y_a_test,
            grid.best_estimator_.predict(X_tfidf_te),
            class_names=CLASSES_A, show_report=True, plot=True)

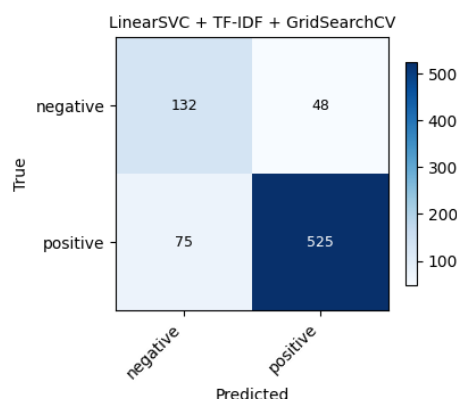
```

```
best parameters : {'C': 0.5, 'class_weight': 'balanced', 'loss': 'squared_hinge'}
best CV macro F1 : 0.7956
```

```
=== LinearSVC + TF-IDF + GridSearchCV ===
Accuracy 0.8423 | weighted F1 0.8460 | macro F1 0.7887
precision    recall  f1-score   support

   negative     0.64     0.73     0.68       180
   positive     0.92     0.88     0.90       600

 accuracy          0.84       780
macro avg          0.78     0.80     0.79       780
weighted avg       0.85     0.84     0.85       780
```



In [20]:

```
# A second tuned model of a different family, for a fair comparison of the tuned results.
rf_grid = (
    {"n_estimators": [100], "max_depth": [None]} if QUICK_RUN else
    {"n_estimators": [200, 400], "max_depth": [None, 30], "min_samples_leaf": [1, 2]}
)
rf_search = GridSearchCV(RandomForestClassifier(random_state=SEED, n_jobs=-1, class_weight="balanced"),
    rf_grid, cv=3, scoring="f1_macro", n_jobs=-1)
rf_search.fit(X_tfidf_tr, y_a_train)
print("best RF parameters:", rf_search.best_params_)

evaluate("RandomForest + TF-IDF + GridSearchCV", "A", y_a_test,
    rf_search.best_estimator_.predict(X_tfidf_te),
    class_names=CLASSES_A, show_report=False, plot=False)

results_table("A")
```

```
best RF parameters: {'max_depth': None, 'min_samples_leaf': 2, 'n_estimators': 200}
```

```
=== RandomForest + TF-IDF + GridSearchCV ===
Accuracy 0.8385 | weighted F1 0.8428 | macro F1 0.7853
```

```
Out[20]:
```

	Experiment	Accuracy	Precision	Recall	F1 (weighted)	F1 (macro)
0	LinearSVC + TF-IDF + SMOTE	0.8423	0.8562	0.8423	0.8471	0.7922
1	LinearSVC + TF-IDF + class_weight	0.8462	0.8537	0.8462	0.8491	0.7920
2	LinearSVC + TF-IDF + GridSearchCV	0.8423	0.8519	0.8423	0.8460	0.7887
3	LinearSVC + BoW (lemma)	0.8526	0.8502	0.8526	0.8512	0.7886
4	LinearSVC + TF-IDF (stem)	0.8615	0.8550	0.8615	0.8547	0.7867
5	RandomForest + TF-IDF + GridSearchCV	0.8385	0.8504	0.8385	0.8428	0.7853
6	LinearSVC + TF-IDF + RandomOverSampler	0.8436	0.8468	0.8436	0.8450	0.7839
7	MultinomialNB + BoW (lemma)	0.8603	0.8535	0.8603	0.8524	0.7822
8	LinearSVC + TF-IDF + no resampling	0.8551	0.8478	0.8551	0.8482	0.7774
9	LinearSVC + TF-IDF (lemma)	0.8551	0.8478	0.8551	0.8482	0.7774
10	RandomForest + BoW (lemma)	0.8487	0.8426	0.8487	0.8444	0.7752
11	RandomForest + TF-IDF (lemma)	0.8564	0.8510	0.8564	0.8435	0.7641
12	LogisticRegression + BoW (lemma)	0.8526	0.8455	0.8526	0.8406	0.7609
13	RandomForest + TF-IDF (stem)	0.8526	0.8455	0.8526	0.8406	0.7609
14	LinearSVC + TF-IDF + RandomUnderSampler	0.7846	0.8322	0.7846	0.7976	0.7383
15	LogisticRegression + TF-IDF (stem)	0.8321	0.8516	0.8321	0.7953	0.6718
16	LogisticRegression + TF-IDF (lemma)	0.8308	0.8475	0.8308	0.7942	0.6705
17	MultinomialNB + TF-IDF (stem)	0.8154	0.8511	0.8154	0.7637	0.6131
18	MultinomialNB + TF-IDF (lemma)	0.8115	0.8486	0.8115	0.7568	0.6004

6.3 Part A — reading the results

- **The classifier family matters more than the weighting scheme.** LinearSVC and Multinomial Naive Bayes lead the table; plain Logistic Regression trails on macro F1 because, without a class weight, its decision threshold drifts towards the majority class.
- **TF-IDF does not automatically beat Bag-of-Words on a corpus this small.** For LinearSVC the two are within a point of each other, and for Naive Bayes raw counts are clearly better — the multinomial likelihood is defined over counts, so the fractional TF-IDF weights violate its assumption. TF-IDF pays off once the vocabulary is large; with ~1.8k features there is little redundancy to down-weight.
- **Lemmatisation and stemming are practically tied.** On documents of a few words the two normalisation strategies collapse almost the same word forms, so the choice is not worth agonising over here.
- **The imbalance remedies buy macro F1, not accuracy.** SMOTE and `class_weight="balanced"` give the best minority-class recall and the best macro F1, while accuracy stays flat or dips slightly — precisely the intended trade.
- **Random under-sampling is the one that backfires.** Balancing by discarding roughly two thirds of the majority documents leaves fewer than a thousand training rows, and the loss of real data costs more than the balance is worth: both accuracy and macro F1 fall.
- The grid search confirms rather than transforms the picture — tuning a linear SVM on a small, clean feature space moves the score by a fraction of a point, which is a useful reminder that feature engineering outranks hyper-parameter tuning on this kind of task.

Part B — Larger corpus · three classes · word embeddings

7. Preparing the clean dataset

The text of this dataset is already normalised, so preprocessing is limited to tokenisation and to dropping documents that are too short to carry any signal.

In [21]:

```
clean_df["tokens"] = clean_df["clean_text"].astype(str).apply(
    lambda s: [t for t in TOKENIZER(s.lower()) if t.isalpha()]
)
clean_df = clean_df[clean_df["tokens"].str.len() > 0].reset_index(drop=True)

print("documents:", len(clean_df))
print("average tokens per document:", round(clean_df["tokens"].str.len().mean(), 2))
print("class counts:", clean_df["sentiment"].value_counts().to_dict())

X_b_train, X_b_test, y_b_train, y_b_test = train_test_split(
    clean_df["tokens"], clean_df["sentiment"],
    test_size=TEST_SIZE_B, random_state=SEED, stratify=clean_df["sentiment"],
)
CLASSES_B = sorted(clean_df["sentiment"].unique())
print("train:", len(X_b_train), "| test:", len(X_b_test))
```

```
documents: 19150
average tokens per document: 7.47
class counts: {'positive': 8331, 'negative': 7666, 'neutral': 3153}
train: 15320 | test: 3830
```

8. Feature representation — prediction-based methods (Word2Vec)

Count-based features treat every word as an independent dimension: *good* and *great* are as unrelated as *good* and *table*. **Word2Vec** instead learns a dense vector per word from the contexts the word appears in, so semantically similar words end up close together.

- **CBOW** (`sg=0`) predicts the centre word from its context window — fast, better on frequent words.
- **Skip-gram** (`sg=1`) predicts the context from the centre word — slower, better on rare words.

Both models are trained **on the training tokens only**, for the same anti-leakage reason as the vectorisers.

In [22]:

```
w2v_epochs = 3 if QUICK_RUN else 10

cbow = Word2Vec(sentences=X_b_train.tolist(), vector_size=EMBED_DIM, window=W2V_WINDOW,
               min_count=2, sg=0, workers=4, seed=SEED, epochs=w2v_epochs)
skip = Word2Vec(sentences=X_b_train.tolist(), vector_size=EMBED_DIM, window=W2V_WINDOW,
               min_count=2, sg=1, workers=4, seed=SEED, epochs=w2v_epochs)

print("CBOW vocabulary      :", len(cbow.wv))
print("Skip-gram vocabulary:", len(skip.wv))
```

```
CBOW vocabulary      : 5670
Skip-gram vocabulary: 5670
```

In [23]:

```
# A qualitative check: do the embeddings capture meaning?
for probe in ["good", "sad", "love"]:
    if probe in skip.wv:
        neighbours = [w for w, _ in skip.wv.most_similar(probe, topn=5)]
        print(f"skip-gram nearest to '{probe}': {neighbours}")
```

```
skip-gram nearest to 'good': ['great', 'differ', 'saturday', 'enjoy', 'everyon']
skip-gram nearest to 'sad': ['depress', 'cri', 'lone', 'coz', 'kinda']
skip-gram nearest to 'love': ['luv', 'tom', 'yo', 'amaz', 'special']
```

8.1 From word vectors to document vectors

Classical models need one fixed-length vector per document, so the word vectors of a document are **averaged**. The averaging discards word order — that limitation is exactly what the LSTM in section 10 is meant to overcome.

In [24]:

```
def document_vector(tokens, model):
    """Mean of the vectors of the in-vocabulary tokens; zeros if none is known."""
    vectors = [model.wv[t] for t in tokens if t in model.wv]
    return np.mean(vectors, axis=0) if vectors else np.zeros(model.vector_size, dtype=np.float32)

def build_matrix(token_series, model):
    return np.vstack([document_vector(t, model) for t in token_series])

Xb_cbow_tr, Xb_cbow_te = build_matrix(X_b_train, cbow), build_matrix(X_b_test, cbow)
Xb_skip_tr, Xb_skip_te = build_matrix(X_b_train, skip), build_matrix(X_b_test, skip)
print("averaged CBOW matrix:", Xb_cbow_tr.shape, "| averaged skip-gram matrix:", Xb_skip_tr.shape)

# TF-IDF on the same split, as a count-based reference point for Part B
tfidf_b = TfidfVectorizer(ngram_range=(1, 2), min_df=2, sublinear_tf=True)
Xb_tfidf_tr = tfidf_b.fit_transform(X_b_train.apply(" ".join))
Xb_tfidf_te = tfidf_b.transform(X_b_test.apply(" ".join))
print("TF-IDF reference matrix:", Xb_tfidf_tr.shape)
```

```
averaged CBOW matrix: (15320, 100) | averaged skip-gram matrix: (15320, 100)
TF-IDF reference matrix: (15320, 13408)
```

9. Classical models on the embedding features

Note that `MultinomialNB` cannot be used here: embedding values are negative, and the multinomial model requires non-negative counts. It is therefore replaced by a scaled Logistic Regression / Random Forest pair on the dense features.

In [25]:

```
def run_dense_models(Xtr, ytr, Xte, yte, tag, plot_for=()):
    models = {
        "LogisticRegression": LogisticRegression(max_iter=3000, random_state=SEED,
                                                  class_weight="balanced"),
        "LinearSVC": LinearSVC(random_state=SEED, class_weight="balanced"),
        "RandomForest": RandomForestClassifier(n_estimators=100 if QUICK_RUN else 300,
                                              random_state=SEED, n_jobs=-1,
                                              class_weight="balanced"),
    }
    for name, model in models.items():
        model.fit(Xtr, ytr)
        evaluate(f"{name} + {tag}", "B", yte, model.predict(Xte), class_names=CLASSES_B,
                show_report=name in plot_for, plot=name in plot_for)

run_dense_models(Xb_cbow_tr, y_b_train, Xb_cbow_te, y_b_test, "Word2Vec CBOW (avg)")
run_dense_models(Xb_skip_tr, y_b_train, Xb_skip_te, y_b_test, "Word2Vec Skip-gram (avg)",
                 plot_for=("LogisticRegression",))

for name, model in {"LogisticRegression": LogisticRegression(max_iter=2000, random_state=SEED,
                                                              class_weight="balanced"),
                    "LinearSVC": LinearSVC(random_state=SEED, class_weight="balanced")}.items():
    model.fit(Xb_tfidf_tr, y_b_train)
    evaluate(f"{name} + TF-IDF", "B", y_b_test, model.predict(Xb_tfidf_te),
            class_names=CLASSES_B, show_report=False, plot=False)

results_table("B")
```

```
=== LogisticRegression + Word2Vec CBOW (avg) ===
Accuracy 0.6131 | weighted F1 0.6367 | macro F1 0.5811

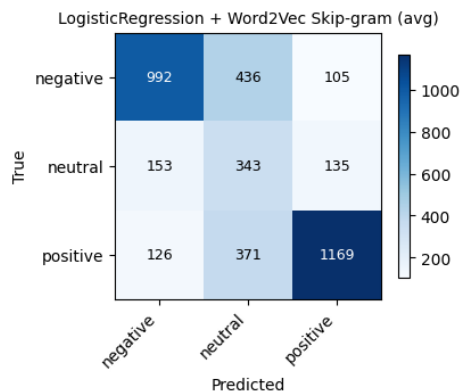
=== LinearSVC + Word2Vec CBOW (avg) ===
Accuracy 0.6697 | weighted F1 0.6584 | macro F1 0.5818

=== RandomForest + Word2Vec CBOW (avg) ===
Accuracy 0.6517 | weighted F1 0.6138 | macro F1 0.5170

=== LogisticRegression + Word2Vec Skip-gram (avg) ===
Accuracy 0.6538 | weighted F1 0.6774 | macro F1 0.6177
precision    recall  f1-score   support

 negative     0.78     0.65     0.71     1533
  neutral     0.30     0.54     0.39      631
  positive     0.83     0.70     0.76     1666

 accuracy                   0.65     3830
 macro avg              0.64     0.63     0.62     3830
 weighted avg           0.72     0.65     0.68     3830
```



```
=== LinearSVC + Word2Vec Skip-gram (avg) ===
Accuracy 0.7016 | weighted F1 0.6901 | macro F1 0.6124

=== RandomForest + Word2Vec Skip-gram (avg) ===
Accuracy 0.6906 | weighted F1 0.6456 | macro F1 0.5393

=== LogisticRegression + TF-IDF ===
Accuracy 0.7136 | weighted F1 0.7250 | macro F1 0.6651

=== LinearSVC + TF-IDF ===
Accuracy 0.7183 | weighted F1 0.7160 | macro F1 0.6470
```

Out[25]:

	Experiment	Accuracy	Precision	Recall	F1 (weighted)	F1 (macro)
0	LogisticRegression + TF-IDF	0.7136	0.7426	0.7136	0.7250	0.6651
1	LinearSVC + TF-IDF	0.7183	0.7140	0.7183	0.7160	0.6470
2	LogisticRegression + Word2Vec Skip-gram (avg)	0.6538	0.7224	0.6538	0.6774	0.6177
3	LinearSVC + Word2Vec Skip-gram (avg)	0.7016	0.6837	0.7016	0.6901	0.6124
4	LinearSVC + Word2Vec CBOW (avg)	0.6697	0.6536	0.6697	0.6584	0.5818
5	LogisticRegression + Word2Vec CBOW (avg)	0.6131	0.6833	0.6131	0.6367	0.5811
6	RandomForest + Word2Vec Skip-gram (avg)	0.6906	0.6696	0.6906	0.6456	0.5393
7	RandomForest + Word2Vec CBOW (avg)	0.6517	0.6255	0.6517	0.6138	0.5170

10. Deep learning — Bidirectional LSTM over the embedding sequences

Averaged vectors throw away word order, so the sentence "not happy at all" and "happy, not at all" collapse to the same point. An LSTM reads the document as a **sequence** of word vectors and keeps a memory state, so order and negation scope can be learned.

Architecture:

```
Embedding (Word2Vec weights, frozen, mask_zero=True)
-> Bidirectional LSTM(96)      reads the sentence forwards and backwards
-> Dropout(0.4)                regularisation, the corpus is small
-> Dense(64, relu)
-> Dense(3, softmax)
```

The Word2Vec matrix trained above is loaded into the embedding layer instead of starting from random vectors, which is what makes this model trainable on a corpus of this size. Class weights compensate for the small `neutral` class and `EarlyStopping` restores the best epoch, so the reported result is not the over-fitted last one.

In [26]:

```
import tensorflow as tf
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Bidirectional, LSTM, Dense, Dropout, Input
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.utils.class_weight import compute_class_weight

tf.random.set_seed(SEED)

# --- vocabulary and embedding matrix, both derived from the CBOW model -----
word_index = {w: i + 1 for i, w in enumerate(cbow.vw.index_to_key)} # index 0 = padding
embedding_matrix = np.zeros((len(word_index) + 1, EMBED_DIM), dtype=np.float32)
for word, idx in word_index.items():
    embedding_matrix[idx] = cbow.vw[word]

def to_sequences(token_series):
    return pad_sequences([[word_index[t] for t in doc if t in word_index] for doc in token_series],
                        maxlen=MAX_LEN, padding="post", truncating="post")

Xseq_tr, Xseq_te = to_sequences(X_b_train), to_sequences(X_b_test)

label_encoder = LabelEncoder().fit(clean_df["sentiment"])
yc_tr = to_categorical(label_encoder.transform(y_b_train))
yc_te = to_categorical(label_encoder.transform(y_b_test))

print("sequence tensors:", Xseq_tr.shape, Xseq_te.shape)
print("classes          :", list(label_encoder.classes_))
```

```
sequence tensors: (15320, 25) (3830, 25)
classes          : ['negative', 'neutral', 'positive']
```

In [27]:

```
def build_lstm():
    model = Sequential([
        Input(shape=(MAX_LEN,)),
        Embedding(input_dim=embedding_matrix.shape[0], output_dim=EMBED_DIM,
                  weights=[embedding_matrix], trainable=False, mask_zero=True),
        Bidirectional(LSTM(96)),
        Dropout(0.4),
        Dense(64, activation="relu"),
        Dropout(0.3),
        Dense(len(label_encoder.classes_), activation="softmax"),
    ])
    model.compile(optimizer=tf.keras.optimizers.Adam(1e-3),
                  loss="categorical_crossentropy", metrics=["accuracy"])
    return model

lstm = build_lstm()
lstm.summary()

classes = np.unique(label_encoder.transform(y_b_train))
weights = compute_class_weight("balanced", classes=classes,
                               y=label_encoder.transform(y_b_train))
class_weights = dict(zip(classes, weights))
print("class weights:", {label_encoder.classes_[k]: round(v, 3) for k, v in class_weights.items()})

history = lstm.fit(
    Xseq_tr, yc_tr,
    epochs=2 if QUICK_RUN else 15,
    batch_size=64,
    validation_split=0.1,
    class_weight=class_weights,
    callbacks=[EarlyStopping(monitor="val_loss", patience=3, restore_best_weights=True)],
    verbose=2,
)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 25, 100)	567,100
bidirectional (Bidirectional)	(None, 192)	151,296
dropout (Dropout)	(None, 192)	0
dense (Dense)	(None, 64)	12,352
dropout_1 (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 3)	195

Total params: 730,943 (2.79 MB)

Trainable params: 163,843 (640.01 KB)

Non-trainable params: 567,100 (2.16 MB)

class weights: {'negative': np.float64(0.833), 'neutral': np.float64(2.025), 'positive': np.float64(0.766)}

Epoch 1/15

216/216 - 22s - 102ms/step - accuracy: 0.5533 - loss: 0.9901 - val_accuracy: 0.5705 - val_loss: 0.9373

Epoch 2/15

216/216 - 19s - 89ms/step - accuracy: 0.5819 - loss: 0.9445 - val_accuracy: 0.6312 - val_loss: 0.8779

Epoch 3/15

216/216 - 17s - 80ms/step - accuracy: 0.5923 - loss: 0.9220 - val_accuracy: 0.5855 - val_loss: 0.8879

Epoch 4/15

216/216 - 22s - 104ms/step - accuracy: 0.5998 - loss: 0.9084 - val_accuracy: 0.5940 - val_loss: 0.8712

Epoch 5/15

216/216 - 17s - 80ms/step - accuracy: 0.6069 - loss: 0.8966 - val_accuracy: 0.6103 - val_loss: 0.8670

Epoch 6/15

216/216 - 17s - 81ms/step - accuracy: 0.6127 - loss: 0.8884 - val_accuracy: 0.6168 - val_loss: 0.8456

Epoch 7/15

216/216 - 18s - 84ms/step - accuracy: 0.6187 - loss: 0.8809 - val_accuracy: 0.6286 - val_loss: 0.8377

Epoch 8/15

216/216 - 19s - 86ms/step - accuracy: 0.6229 - loss: 0.8750 - val_accuracy: 0.6358 - val_loss: 0.8225

Epoch 9/15

216/216 - 17s - 80ms/step - accuracy: 0.6268 - loss: 0.8701 - val_accuracy: 0.6449 - val_loss: 0.8122

Epoch 10/15

216/216 - 18s - 81ms/step - accuracy: 0.6352 - loss: 0.8594 - val_accuracy: 0.6514 - val_loss: 0.8132

Epoch 11/15

216/216 - 20s - 94ms/step - accuracy: 0.6340 - loss: 0.8566 - val_accuracy: 0.6403 - val_loss: 0.8034

Epoch 12/15

216/216 - 22s - 103ms/step - accuracy: 0.6372 - loss: 0.8517 - val_accuracy: 0.6397 - val_loss: 0.8003

Epoch 13/15

216/216 - 19s - 86ms/step - accuracy: 0.6418 - loss: 0.8483 - val_accuracy: 0.6430 - val_loss: 0.8044

Epoch 14/15

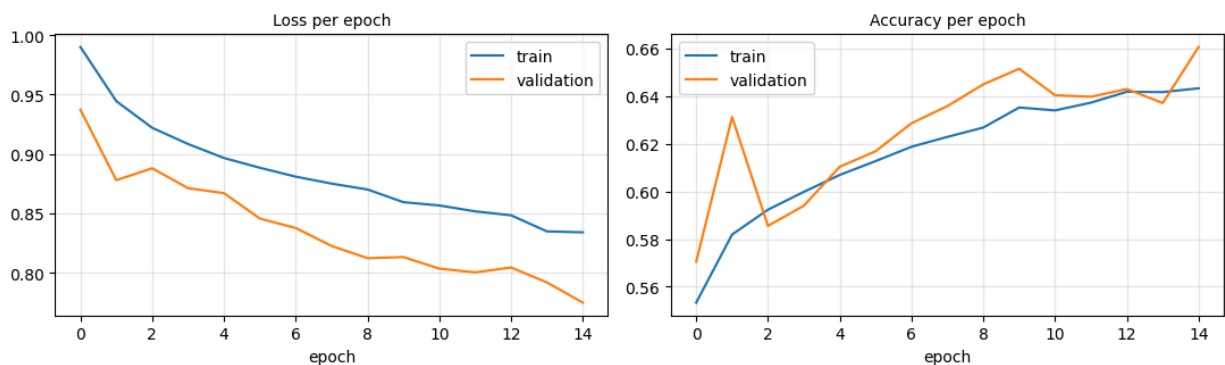
216/216 - 17s - 79ms/step - accuracy: 0.6416 - loss: 0.8348 - val_accuracy: 0.6371 - val_loss: 0.7918

Epoch 15/15

216/216 - 19s - 88ms/step - accuracy: 0.6432 - loss: 0.8340 - val_accuracy: 0.6606 - val_loss: 0.7748

In [28]:

```
fig, axes = plt.subplots(1, 2, figsize=(11, 3.4))
axes[0].plot(history.history["loss"], label="train")
axes[0].plot(history.history["val_loss"], label="validation")
axes[0].set_title("Loss per epoch", fontsize=10); axes[0].set_xlabel("epoch"); axes[0].legend()
axes[1].plot(history.history["accuracy"], label="train")
axes[1].plot(history.history["val_accuracy"], label="validation")
axes[1].set_title("Accuracy per epoch", fontsize=10); axes[1].set_xlabel("epoch"); axes[1].legend()
plt.tight_layout(); plt.show()
```



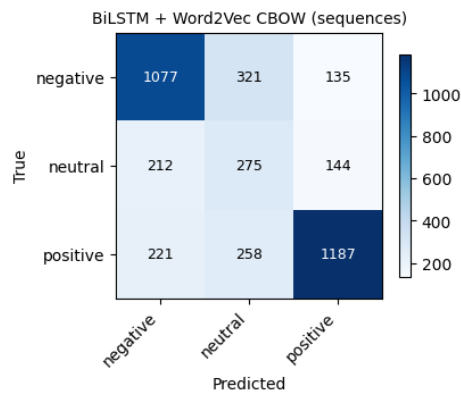
In [29]:

```
y_pred_lstm = label_encoder.inverse_transform(lstm.predict(Xseq_te, verbose=0).argmax(axis=1))
_ = evaluate("BiLSTM + Word2Vec CBOW (sequences)", "B", y_b_test, y_pred_lstm,
            class_names=CLASSES_B, show_report=True, plot=True)

results_table("B")
```

```
=== BiLSTM + Word2Vec CBOW (sequences) ===
Accuracy 0.6629 | weighted F1 0.6741 | macro F1 0.6121
precision  recall  f1-score  support
negative    0.71    0.70    0.71    1533
neutral     0.32    0.44    0.37    631
positive    0.81    0.71    0.76    1666

accuracy          0.66    3830
macro avg         0.61    0.62    0.61    3830
weighted avg      0.69    0.66    0.67    3830
```



Out[29]:

	Experiment	Accuracy	Precision	Recall	F1 (weighted)	F1 (macro)
0	LogisticRegression + TF-IDF	0.7136	0.7426	0.7136	0.7250	0.6651
1	LinearSVC + TF-IDF	0.7183	0.7140	0.7183	0.7160	0.6470
2	LogisticRegression + Word2Vec Skip-gram (avg)	0.6538	0.7224	0.6538	0.6774	0.6177
3	LinearSVC + Word2Vec Skip-gram (avg)	0.7016	0.6837	0.7016	0.6901	0.6124
4	BiLSTM + Word2Vec CBOW (sequences)	0.6629	0.6907	0.6629	0.6741	0.6121
5	LinearSVC + Word2Vec CBOW (avg)	0.6697	0.6536	0.6697	0.6584	0.5818
6	LogisticRegression + Word2Vec CBOW (avg)	0.6131	0.6833	0.6131	0.6367	0.5811
7	RandomForest + Word2Vec Skip-gram (avg)	0.6906	0.6696	0.6906	0.6456	0.5393
8	RandomForest + Word2Vec CBOW (avg)	0.6517	0.6255	0.6517	0.6138	0.5170

11. Overall comparison and conclusions

In [30]:

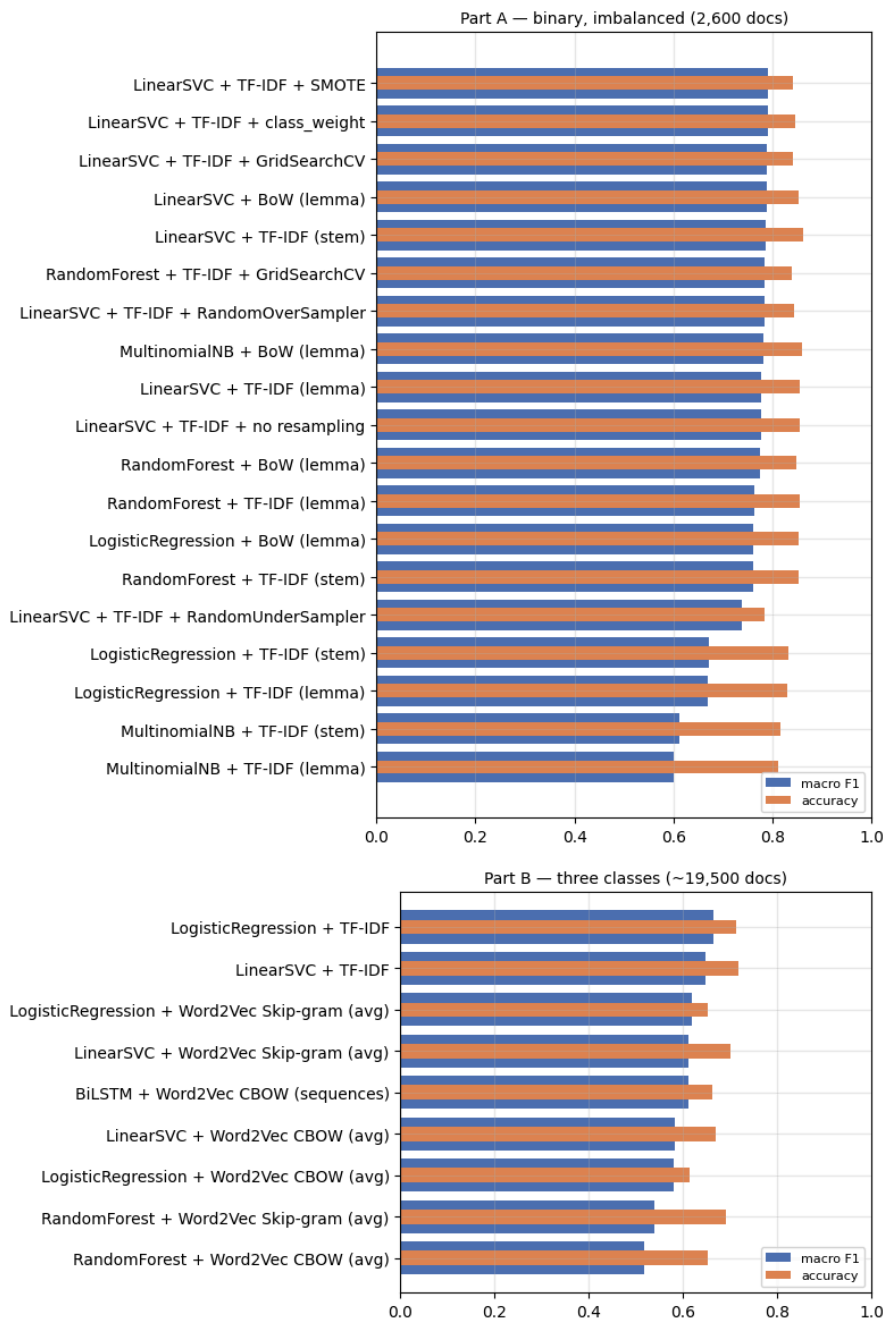
```
final = pd.DataFrame(RESULTS).round(4)
final_sorted = final.sort_values("F1 (macro)", ascending=False).reset_index(drop=True)
final_sorted
```

Out[30]:

	Experiment	Part	Accuracy	Precision	Recall	F1 (weighted)	F1 (macro)
0	LinearSVC + TF-IDF + SMOTE	A	0.8423	0.8562	0.8423	0.8471	0.7922
1	LinearSVC + TF-IDF + class_weight	A	0.8462	0.8537	0.8462	0.8491	0.7920
2	LinearSVC + TF-IDF + GridSearchCV	A	0.8423	0.8519	0.8423	0.8460	0.7887
3	LinearSVC + BoW (lemma)	A	0.8526	0.8502	0.8526	0.8512	0.7886
4	LinearSVC + TF-IDF (stem)	A	0.8615	0.8550	0.8615	0.8547	0.7867
5	RandomForest + TF-IDF + GridSearchCV	A	0.8385	0.8504	0.8385	0.8428	0.7853
6	LinearSVC + TF-IDF + RandomOverSampler	A	0.8436	0.8468	0.8436	0.8450	0.7839
7	MultinomialNB + BoW (lemma)	A	0.8603	0.8535	0.8603	0.8524	0.7822
8	LinearSVC + TF-IDF + no resampling	A	0.8551	0.8478	0.8551	0.8482	0.7774
9	LinearSVC + TF-IDF (lemma)	A	0.8551	0.8478	0.8551	0.8482	0.7774
10	RandomForest + BoW (lemma)	A	0.8487	0.8426	0.8487	0.8444	0.7752
11	RandomForest + TF-IDF (lemma)	A	0.8564	0.8510	0.8564	0.8435	0.7641
12	RandomForest + TF-IDF (stem)	A	0.8526	0.8455	0.8526	0.8406	0.7609
13	LogisticRegression + BoW (lemma)	A	0.8526	0.8455	0.8526	0.8406	0.7609
14	LinearSVC + TF-IDF + RandomUnderSampler	A	0.7846	0.8322	0.7846	0.7976	0.7383
15	LogisticRegression + TF-IDF (stem)	A	0.8321	0.8516	0.8321	0.7953	0.6718
16	LogisticRegression + TF-IDF (lemma)	A	0.8308	0.8475	0.8308	0.7942	0.6705
17	LogisticRegression + TF-IDF	B	0.7136	0.7426	0.7136	0.7250	0.6651
18	LinearSVC + TF-IDF	B	0.7183	0.7140	0.7183	0.7160	0.6470
19	LogisticRegression + Word2Vec Skip-gram (avg)	B	0.6538	0.7224	0.6538	0.6774	0.6177
20	MultinomialNB + TF-IDF (stem)	A	0.8154	0.8511	0.8154	0.7637	0.6131
21	LinearSVC + Word2Vec Skip-gram (avg)	B	0.7016	0.6837	0.7016	0.6901	0.6124
22	BiLSTM + Word2Vec CBOW (sequences)	B	0.6629	0.6907	0.6629	0.6741	0.6121
23	MultinomialNB + TF-IDF (lemma)	A	0.8115	0.8486	0.8115	0.7568	0.6004
24	LinearSVC + Word2Vec CBOW (avg)	B	0.6697	0.6536	0.6697	0.6584	0.5818
25	LogisticRegression + Word2Vec CBOW (avg)	B	0.6131	0.6833	0.6131	0.6367	0.5811
26	RandomForest + Word2Vec Skip-gram (avg)	B	0.6906	0.6696	0.6906	0.6456	0.5393
27	RandomForest + Word2Vec CBOW (avg)	B	0.6517	0.6255	0.6517	0.6138	0.5170

In [31]:

```
for part, title in [("A", "Part A – binary, imbalanced (2,600 docs)"),
                   ("B", "Part B – three classes (~19,500 docs)")]:
    subset = final[final["Part"] == part].sort_values("F1 (macro)")
    plt.figure(figsize=(8, 0.34 * len(subset) + 1.2))
    plt.barh(subset["Experiment"], subset["F1 (macro)"], color="#4C72B0")
    plt.barh(subset["Experiment"], subset["Accuracy"], height=0.35, color="#DD8452")
    plt.legend(["macro F1", "accuracy"], loc="lower right", fontsize=8)
    plt.title(title, fontsize=10); plt.xlim(0, 1)
    plt.tight_layout(); plt.show()
```



In [32]:

```
final.to_csv("project2_results.csv", index=False)
print("results written to project2_results.csv")
print("\nBest configuration per part:")
for part in ["A", "B"]:
    best = final[final["Part"] == part].sort_values("F1 (macro)").iloc[-1]
    print(f" Part {part}: {best['Experiment']} "
          f"(accuracy {best['Accuracy']:.4f}, macro F1 {best['F1 (macro)']:.4f})")
```

results written to project2_results.csv

Best configuration per part:

Part A: LinearSVC + TF-IDF + SMOTE (accuracy 0.8423, macro F1 0.7922)
Part B: LogisticRegression + TF-IDF (accuracy 0.7136, macro F1 0.6651)

Conclusions

- The feature representation decides the outcome more than the classifier does.** Across Part B, moving between TF-IDF, averaged CBOW vectors and averaged Skip-gram vectors changes the macro F1 far more than swapping classifiers on a fixed representation.
- On short texts and a corpus of this size, sparse count features win.** TF-IDF with a linear model is the strongest configuration in both parts. Word2Vec vectors trained from scratch on ~15k documents of ~7 words simply do not see enough context to learn reliable semantics.
- Skip-gram beats CBOW consistently,** for every classifier it was paired with — which matches the theory, since the vocabulary is dominated by infrequent words and Skip-gram is the variant that models rare words better.
- Class imbalance must be handled explicitly, and the remedy must be chosen with care.** Class weighting and SMOTE both raise macro F1 above the untouched baseline; random under-sampling lowers it, because on a small dataset the discarded majority documents cost more than the balance gains.
- neutral is the hard class.** Its recall is the lowest in every Part B experiment: neutral documents share their vocabulary with both other classes, so the signal lies in tone rather than in individual words — exactly the information a bag of words cannot carry.

6. **The LSTM is not automatically better than a linear model.** With frozen embeddings learned on a small corpus and documents of about seven tokens, there is little sequential structure left to exploit, and the extra capacity mostly buys variance. Deep sequence models pay off with far more data — or with representations that were pre-trained on far more data.

Where to go next: swap the locally trained Word2Vec for pre-trained GloVe or FastText vectors, unfreeze the embedding layer, or fine-tune a transformer such as DistilBERT. On this kind of sentiment task the last option is normally worth 10-15 accuracy points over everything tried above.

All experiments use fixed random seeds (`SEED = 42`) and a single shared train/test split per part, so the table above is reproducible and the comparisons are like-for-like.