nlp_bow_tfidf_embeddings_lab

October 4, 2025

1 Lab: Comparing Bag-of-Words, TF-IDF, and Embeddings

https://github.com/iankelk/cs89b-sections

Task: News outlet classification (+ a unified semantic search demo)

Dataset: `ik-news.csv` with columns `id, title, publication, author, date, year, month, url, content` **Text column:** `content` **Label column:** `publication`

1.1 What we'll show today (60 min)

1. **Prep the dataset**: quick schema checks, light normalization (`text_norm`), and label distribution.
2. **Leakage guard**: remove source cues (publisher names, URLs, datelines/bylines) and dedupe/group-split to make evaluation fair.
3. **Bag-of-Words (BoW)**: unigrams & bigrams; explore frequent terms and train a simple linear classifier.
4. **TF-IDF**: swap raw counts for TF-IDF weights; experiment with sublinear TF scaling, normalization, and character n-grams.
5. **Embeddings (CBOW)**: train local word2vec embeddings → build document vectors via mean and TF-IDF–weighted pooling → compare performance.
6. **Pretrained embeddings (GoogleNews W2V)**: integrate off-the-shelf 300-d vectors and benchmark them against the local CBOW model.
7. **Unified benchmark**: evaluate all models side-by-side (BoW, TF-IDF, char n-grams, CBOW, pretrained W2V) with accuracy and macro-F1.
8. **Semantic search**: build a unified retrieval demo using
   - **TF-IDF (lexical)**: keyword-based similarity
   - **Local CBOW (semantic)**: meaning-aware matches from trained vectors
   - **Pretrained W2V (semantic++)**: high-quality retrieval using GoogleNews embeddings

Goals

- See how **feature choices** (counts → TF-IDF → embeddings) shape what the model learns.
- Learn to **bridge symbolic and neural NLP** using TF-IDF-weighted embeddings.
- Understand how to **move from lexical matching to semantic retrieval**.

- Build intuition: BoW/TF-IDF excel at **style and phrasing**, embeddings capture **meaning and context**.

1.2 0) Setup

[ ]:
```
# If needed (e.g., on Colab), uncomment:
# !pip install scikit-learn pandas numpy matplotlib gensim
```

[ ]:
```
import pandas as pd
import numpy as np
import re
from collections import Counter
from itertools import chain

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer,␣
↪TfidfTransformer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, f1_score,␣
↪confusion_matrix
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.model_selection import GroupShuffleSplit
from sklearn.pipeline import make_pipeline
from sklearn.svm import LinearSVC
from gensim import downloader as api

from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
STOPWORDS = set(ENGLISH_STOP_WORDS)
STOPWORDS.update({'com', 'http', 'https', 'www'})
STOPWORDS = sorted(STOPWORDS)

import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (7, 5)
```

1.3 Part 1 — Dataset prep (what & why)

Goal: Make sure the corpus is usable and the label is sane before we vectorize.

What we do - **Load** `ik-news.csv` and confirm we have the expected columns. - **Pick** our working fields:
- `text_col = "content"` (article body)
- `label_col = "publication"` (Fox News / New York Times / Reuters) - **Minimal normalize** the text to `text_norm` via **lowercasing + strip** (we do *not* remove stopwords here). - **Sanity checks**: - Drop rows where `content` is missing. - Look at **document length** stats (very short docs are often noise). - Inspect **label distribution** (class imbalance affects accuracy).

Why keep stopwords in prep?
Stopwords, numbers, and punctuation decisions are **feature-engineering choices** that we want

to toggle per method (BoW vs TF-IDF vs embeddings). If we delete them here, we can't A/B those choices later. So: prep = light & reversible; filtering happens in vectorizers.

[ ]:
```
df = pd.read_csv('ik-news.csv')
df.shape
```

[ ]:
```
(1139, 9)
```

[ ]:
```
text_col = 'content'
label_col = 'publication'
df[[text_col, label_col]].head(3)
```

[ ]:

	content	publication
0	MONTAGUE, Mass. – Think of all the dogs out…	New York Times
1	WASHINGTON – Gov. Terry McAuliffe of Virgin…	New York Times
2	WASHINGTON – The Supreme Court on Wednesday…	New York Times

[ ]:
```
# Compute % of rows where the text column is null (missing)
null_rate = df[text_col].isna().mean()
print(f"Null rate in text_col: {null_rate:.3f}")

# Drop rows with missing text so vectorizers don't crash
df = df[df[text_col].notna()].copy()

# Quick sanity check on document lengths (characters)
doc_lengths = df[text_col].astype(str).str.len()
print(doc_lengths.describe())    # shows count/mean/std/min/percentiles/max

# If a label column exists, show top label counts (helps spot imbalance)
if label_col in df.columns:
    print("\nLabel distribution (top 10):")
    print(df[label_col].value_counts().head(10))
```

```
Null rate in text_col: 0.000
count     1139.000000
mean      4839.935031
std       2685.199097
min        109.000000
25%       2979.000000
50%       4439.000000
75%       6256.500000
max      27119.000000
Name: content, dtype: float64

Label distribution (top 10):
publication
New York Times    478
Fox News          360
```

```
Reuters            301
Name: count, dtype: int64
```

What "normalize" means here: * Lowercasing → "Apple" and "apple" become the same token. * Strip → removes stray spaces at the start/end.

This is intentionally lightweight; we leave stopword removal, token pattern choices, etc. to the vectorizers, so we can A/B those per method.

[ ]:
```
def normalize(s: str) -> str:
    # minimal normalization: make lowercase and trim outer whitespace
    return str(s).lower().strip()

# Create a working text column used by all later steps
df['text_norm'] = df[text_col].apply(normalize)
df['text_norm'].head(3)
```

[ ]:
```
0    montague, mass.  -   think of all the dogs out…
1    washington  -   gov. terry mcauliffe of virgin…
2    washington  -   the supreme court on wednesday…
Name: text_norm, dtype: object
```

1.4 Part 2 — Leakage guard (what & why)

Problem: Publisher/source classification can be trivially "solved" if the text contains **explicit source markers** — e.g., "(Reuters)", "nytimes.com", "— Fox News —", or bylines like "Reporting by …". If those tokens cross into your features, models learn the shortcut and inflated scores follow.

What we remove - **URLs & bare domains** (e.g., `foxnews.com`, `nytimes.com`, `reuters.com`) — these often tokenize into junk bigrams like `foxnews com`. - **Publisher variants** with/without spaces/dots (e.g., `fox news`, `foxnews`, `ny times`, `nytimes`), including obvious aliases. - **Datelines/bylines boilerplate** at the top or tail (e.g., `WASHINGTON (Reuters) -`, `Reporting by ...`, `Contributed to this report`).

How we do it - Build a **regex scrubber** that: 1) strips URLs/domains,
2) removes publisher name variants,
3) deletes common dateline/byline shells,
4) collapses whitespace.
- Apply it to **`title` + `content`**, then set `text_norm = text_clean.lower().strip()` so all later vectorizers use the cleaned text. - **Deduplicate exact repeats** (`title` + `content`) to avoid the same wire story in both train and test. - Use a **group-aware split** by `url` or `title` so near-duplicates don't straddle train/test.

Why this matters - After this guard, high scores indicate the model is picking up **style and phrasing patterns**, not literal source names.
- It also makes BoW vs TF-IDF vs embeddings a **fair comparison**: we're testing representations, not leakages.

After training, inspect **top features per class** (for BoW/TF-IDF). If you still see `foxnews`, `nytimes`, or raw domains, tighten the scrub and/or add URL tokens like `com/http/www` to your vectorizer stopwords.

[ ]:

```python
# ---- A) Remove URLs and bare domains (e.g., foxnews.com, nytimes.com, reuters.com) ----
URL_OR_DOMAIN = re.compile(
    r'(https?://\S+|www\.\S+|\b[a-z0-9][a-z0-9-]+(?:\.[a-z0-9-]+)+(?:/\S*)?)',
    flags=re.I
)
def strip_urls_domains(t: str) -> str:
    return URL_OR_DOMAIN.sub(' ', str(t))

# ---- B) Scrub outlet name variants (spaces/dots/hyphens optional) ----
OUTLET_STRONG = re.compile(r"""
\b(
    reuters(?:\s*[\.\-]?\s*com)? |
    fox(?:\s*[-\s]*news(?:\s*channel)?) |
    foxnews(?:\s*[\.\-]?\s*com)? |
    (?:the\s+)?new\s+york\s+times |
    newyorktimes |
    ny\s*times |
    nytimes(?:\s*[\.\-]?\s*com)?
)\b
""", flags=re.I | re.X)

# ---- C) Optional wire-service cues (they had appeared in the top features when I first ran this) ----
WIRE_CUES = re.compile(r"\bassociated\s+press\b|\bap\s*news\b|\bapnews\b", flags=re.I)
REMOVE_WIRE_CUES = True  # set False if you want to keep AP mentions

# ---- D) Dateline/byline shells (handles – or -) + end-of-article bylines ----
DATELINE_PAT = re.compile(r"^\s*[A-Z][A-Z\s\.,'&-]+(?:\(\s*reuters\s*\))?\s*[-
-]\s*", re.I)
BYLINE_PAT   = re.compile(
    r"(reporting by|with reporting by|edited by|editing by|writing by|contributed by|contributed to this report).*$",
    re.I
)

def strict_scrub(title: str, content: str) -> str:
    s = f"{str(title)}. {str(content)}"
    s = strip_urls_domains(s)
    s = OUTLET_STRONG.sub(' ', s)
    if REMOVE_WIRE_CUES:
        s = WIRE_CUES.sub(' ', s)
    s = re.sub(DATELINE_PAT, ' ', s)
    s = re.sub(BYLINE_PAT, ' ', s)
    s = re.sub(r"\s+", " ", s).strip()
    return s
```

```
# ---- Apply once to build working text ----
df['text_clean'] = [
    strict_scrub(t, c) for t, c in zip(df['title'].astype(str), df['content'].
 ↪astype(str))
]
df['text_norm'] = df['text_clean'].map(normalize)

# ---- Deduplicate exact repeats (helps with split leakage) ----
df = df.drop_duplicates(subset=['title', 'content']).copy()

print("After leakage guard - rows:", len(df))
print(df['text_norm'].head(1).item()[:300], "...")
```

```
After leakage guard - rows: 1139
think of all the dogs out there: labradors and poodles and labradoodles huskies
and westies and dogues de bordeaux pit bulls and spaniels and lovable mutts that
go to doggy day care. add them up, all the pet dogs on the planet, and you get
about 250 million. but there are about a billion dogs on ear …
```

What "group-aware split" means: * Instead of randomly splitting rows, we split by groups (URL/title). * All items with the same group value stay together (avoid the same article or its near-duplicate showing up in both train and test). * This gives you a more honest estimate of generalization by reducing accidental data leakage.

[ ]:
```
# === Single source of truth: group-aware train/test split ===
label_col = 'publication' if 'label_col' not in globals() else label_col

# Group by URL; fall back to title when URL is missing
groups = df['url'].fillna(df['title']).astype(str)

gss = GroupShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
train_idx, test_idx = next(gss.split(df, groups=groups))

# Canonical variables used everywhere below
# Reset index to keep lengths aligned when you later convert to arrays
# or build DataFrames (no lingering original indices).
X_train_txt = df.loc[train_idx, 'text_norm'].reset_index(drop=True)
X_test_txt  = df.loc[test_idx,  'text_norm'].reset_index(drop=True)
y_train     = df.loc[train_idx, label_col].reset_index(drop=True)
y_test      = df.loc[test_idx,  label_col].reset_index(drop=True)

print(f"Group-aware split   train={len(X_train_txt)}  test={len(X_test_txt)}")
```

```
Group-aware split   train=911  test=228
```

1.5 3) Manual BoW (hand-built to see the mechanics)

We take a tiny sample of documents, tokenize them with a simple regex, build a vocabulary (word → column index), and manually fill a document–term matrix counting how many times each word appears in each doc. It's BoW from first principles so you can see how simple the mechanics are.

[ ]:
```
# --- Tokenize: split on letters/apostrophes so "don't" -> ["don", "t"]
TOKEN_RE = re.compile(r"[A-Za-z']+")
def tokenize(text: str):
    return TOKEN_RE.findall(text)

# Grab a tiny slice so it's easy to inspect
sample_docs = df['text_norm'].head(6).tolist()
sample_tokens = [tokenize(doc) for doc in sample_docs]

# Build a vocabulary by frequency: most common word gets column 0, etc.
vocab_counts = Counter(chain.from_iterable(sample_tokens))
vocab = {w: i for i, (w, _) in enumerate(vocab_counts.most_common())}
print('Tiny sample vocab size =', len(vocab))

# Allocate a dense doc-term matrix (rows: docs, cols: vocab), then fill counts
X_demo = np.zeros((len(sample_tokens), len(vocab)), dtype=int)
for d, toks in enumerate(sample_tokens):
    for w in toks:
        X_demo[d, vocab[w]] += 1

print('X_demo shape:', X_demo.shape)   # (#docs, #unique_words)
X_demo[:3, :10]                        # peek at first 3 docs / first 10 vocab␣
↪cols
```

```
Tiny sample vocab size = 1882
X_demo shape: (6, 1882)
```

[ ]:
```
array([[106,  54,  66,  50,  73,  53,  41,  80,  18,  49],
       [ 66,  45,  30,  31,  21,  34,  20,   0,  19,   4],
       [ 60,  33,  34,  24,  18,  21,  28,   0,  15,   2]])
```

1.6 4) Scikit-learn BoW (unigrams)

Same idea as the manual build, but using CountVectorizer to tokenize, build a vocabulary, and produce a sparse document–term matrix efficiently for thousands of documents.

We also add filters to keep the vocabulary informative and manageable: * `min_df=2` → Minimum document frequency.

A term must appear in at least 2 documents to be kept.

- This removes extremely rare words, typos, or single-use names that add noise but little predictive power.

- Example: if a word only occurs once in a 1000-article corpus, it probably doesn't generalize well.
- `max_df` (optional) → Maximum document frequency.

A term appearing in too many documents (e.g., >70%) is usually too generic ("said", "news", "the"). Setting `max_df=0.7` tells the vectorizer to drop words that appear in more than 70% of documents, since they don't help distinguish classes

- `stop_words=STOPWORDS` → removes common function words ("the", "and", "of") that rarely carry signal.
- `max_features=20000` → keeps only the top-20k most frequent terms, ensuring the matrix stays a reasonable size.

Together, these parameters balance coverage (keep enough distinctive terms) and noise reduction (drop words that are too rare or too common).

[ ]:
```python
# Configure a unigram (single-word) count vectorizer
vectorizer_uni = CountVectorizer(
    lowercase=True,            # lowercase during vectorization
    ngram_range=(1,1),         # unigrams only
    min_df=2,                  # keep tokens that appear in >= 2 docs
    stop_words=STOPWORDS,      # shared stopword list (keeps mr/ms)
    max_features=20000         # cap vocab to the top 20k features
)

# Fit on the whole corpus text, then transform to a sparse matrix
X_uni = vectorizer_uni.fit_transform(df['text_norm'])
print('X_uni shape (docs × vocab):', X_uni.shape)

# Inspect the most frequent unigrams overall (sum counts per column)
term_counts = np.asarray(X_uni.sum(axis=0)).ravel()
terms = vectorizer_uni.get_feature_names_out()
order = term_counts.argsort()[::-1]

print("Top unigrams:")
for idx in order[:25]:
    print(f"{terms[idx]:<20} {int(term_counts[idx])}")
```

```
X_uni shape (docs × vocab): (1139, 18173)
Top unigrams:
said                 8422
mr                   4813
trump                3330
people               1935
clinton              1904
new                  1578
state                1527
percent              1253
like                 1236
```

```
time                    1120
years                   1062
year                    1060
president               1054
campaign                1038
told                    1035
just                    1027
states                  1013
obama                   960
did                     944
united                  870
party                   821
officials               782
republican              780
world                   750
government              744
```

1.7 5) Scikit-learn BoW (unigrams + bigrams)

Unigrams catch which words occur; bigrams add short phrases/collocations (e.g., mr trump, white house), which often carry outlet style and disambiguate meaning.

[ ]:
```python
# Uni+Bi-gram vectorizer: same filters, now ngram_range=(1,2)
vectorizer_uni_bi = CountVectorizer(
    lowercase=True,
    ngram_range=(1,2),        # include unigrams and bigrams
    min_df=2,
    stop_words=STOPWORDS,
    max_features=30000
)

X_uni_bi = vectorizer_uni_bi.fit_transform(df['text_norm'])
print('X_uni_bi shape (docs × vocab):', X_uni_bi.shape)

# Compare vocabulary sizes
print('Unigram vocab:', len(vectorizer_uni.get_feature_names_out()))
print('Uni+Bi vocab:', len(vectorizer_uni_bi.get_feature_names_out()))

# Look at frequent bigrams (filter terms that contain a space)
terms_ub = vectorizer_uni_bi.get_feature_names_out()
counts_ub = np.asarray(X_uni_bi.sum(axis=0)).ravel()
order_ub = counts_ub.argsort()[::-1]

print("Frequent bigrams:")
n = 40
top_pairs = [(terms_ub[i], int(counts_ub[i])) for i in order_ub[:n] if ' ' in␣
↪terms_ub[i]]
for t, c in top_pairs[:20]:
```

```
    print(f"{t:<30} {c}")
```

```
X_uni_bi shape (docs × vocab): (1139, 30000)
Unigram vocab: 18173
Uni+Bi vocab: 30000
Frequent bigrams:
mr trump                       1323
united states                  736
```

1.8 6) Quick baseline classifier on BoW

We split into train/test (reusing the group-aware split), build BoW(1,2) features, and train a simple Logistic Regression classifier to predict the publication. Then we print overall accuracy and a per-class precision/recall/F1 report.

For sparse, high-dimensional n-gram features, a linear model (LogReg/LinearSVC) is fast, robust, and typically near state-of-the-art for bag-of-words style tasks.

[ ]:

```
# BoW features with unigrams+bigrams (counts). Switch to binary=True if desired.
bow = CountVectorizer(
    lowercase=True,
    ngram_range=(1,2),
    min_df=2,
    max_features=30000,
    stop_words=STOPWORDS,
    token_pattern=r"(?u)\b[a-zA-Z][a-zA-Z'-]{1,}\b"
)
X_train = bow.fit_transform(X_train_txt)
X_test  = bow.transform(X_test_txt)

# Linear classifier on sparse features (strong baseline for text)
clf = LogisticRegression(max_iter=1000)
clf.fit(X_train, y_train)
preds = clf.predict(X_test)

# Evaluation
print('Accuracy:', accuracy_score(y_test, preds))
print(classification_report(y_test, preds))
```

```
Accuracy: 0.8114035087719298
                precision    recall  f1-score   support

      Fox News       0.69      0.84      0.76        61
New York Times       0.99      0.81      0.89       110
       Reuters       0.70      0.79      0.74        57

      accuracy                           0.81       228
     macro avg       0.79      0.81      0.80       228
```

```
weighted avg       0.84      0.81      0.82       228
```

---

2 Part 2: TF-IDF (Term Frequency × Inverse Document Frequency)

2.1 1) Unigram TF-IDF

Instead of raw counts, weight each term by how specific it is to fewer documents.

- TF (term frequency): how often a term appears in a document.
- IDF (inverse document frequency): larger for rare terms, smaller for common ones.
- Combined (TF×IDF) = highlight terms that are frequent in this doc but not everywhere.

Key params here:

- `min_df=2`: drop terms that occur in only 1 doc (likely noise).
- `stop_words=STOPWORDS`: remove common function words.
- `norm='l2'`: per-document L2 normalization (so long docs don't dominate).
- `use_idf=True, smooth_idf=True`: standard TF-IDF with smoothing.

[ ]:
```python
# === Unigram TF-IDF ===
tfidf_uni = TfidfVectorizer(
    lowercase=True,
    ngram_range=(1,1),          # unigrams only
    min_df=2,
    stop_words=STOPWORDS,
    max_features=20000,
    norm='l2',                  # length-normalize each row vector
    use_idf=True, smooth_idf=True
)

X_tfidf_uni = tfidf_uni.fit_transform(df['text_norm'])
print('TF-IDF unigram shape:', X_tfidf_uni.shape)

# Inspect the IDF part: low IDF = common; high IDF = rare
terms = tfidf_uni.get_feature_names_out()
idf = tfidf_uni.idf_
order_low  = idf.argsort()           # most common first
order_high = idf.argsort()[::-1]     # rarest first

print('Common terms (lowest IDF):')
for i in order_low[:15]:
    print(f"{terms[i]:<20} idf={idf[i]:.3f}")

print('\nRare terms (highest IDF):')
for i in order_high[:15]:
```

```
    print(f"{terms[i]:<20} idf={idf[i]:.3f}")
```

```
TF-IDF unigram shape: (1139, 18173)
Common terms (lowest IDF):
said                 idf=1.090
people               idf=1.492
new                  idf=1.584
time                 idf=1.637
like                 idf=1.713
did                  idf=1.738
years                idf=1.742
just                 idf=1.758
year                 idf=1.791
told                 idf=1.795
state                idf=1.840
president            idf=1.869
including            idf=1.903
states               idf=1.966
week                 idf=1.989

Rare terms (highest IDF):
zookeepers           idf=6.940
zora                 idf=6.940
085                  idf=6.940
077                  idf=6.940
0600                 idf=6.940
05                   idf=6.940
04                   idf=6.940
100th                idf=6.940
liberally            idf=6.940
01                   idf=6.940
liang                idf=6.940
élan                 idf=6.940
liaison              idf=6.940
libraries            idf=6.940
zingers              idf=6.940
```

[ ]:

```
for r in df.sample(3, random_state=1).index:
    row = X_tfidf_uni[r].toarray().ravel()
    top_idx = row.argsort()[::-1][:10]
    print(f"\nDoc {r} - top TF-IDF terms:")
    for j in top_idx:
        if row[j] > 0:
            print(f"  {terms[j]:<25} {row[j]:.3f}")
```

```
Doc 833 - top TF-IDF terms:
  veterans                  0.540
```

```
  heroes                    0.268
  fundraiser                0.191
  ptsd                      0.143
  therapy                   0.137
  organization              0.131
  enforcement               0.130
  injuries                  0.124
  team                      0.109
  vets                      0.102

Doc 1123 - top TF-IDF terms:
  cleveland                 0.283
  demonstrators             0.267
  aclu                      0.232
  convention                0.213
  zone                      0.210
  square                    0.209
  organize                  0.186
  protesters                0.161
  protests                  0.161
  event                     0.152

Doc 584 - top TF-IDF terms:
  criminal                  0.355
  clinton                   0.315
  inquiry                   0.267
  probe                     0.263
  earnest                   0.222
  email                     0.186
  justice                   0.181
  investigation             0.171
  conducting                0.161
  department                0.151
```

2.2 2) TF-IDF (unigrams + bigrams)

Bigrams capture short phrases/collocations (e.g., `mr trump`, `white house`) that unigrams can't. This often helps for style/source tasks, where wording patterns matter.

[ ]:
```
# === TF-IDF with unigrams + bigrams ===
tfidf_uni_bi = TfidfVectorizer(
    lowercase=True,
    ngram_range=(1,2),          # unigrams + bigrams
    min_df=2,
    stop_words=STOPWORDS,
    max_features=30000,
    norm='l2', use_idf=True, smooth_idf=True
)
```

```python
X_tfidf_uni_bi = tfidf_uni_bi.fit_transform(df['text_norm'])
print('TF-IDF (1,2) shape:', X_tfidf_uni_bi.shape)

# Show high-weight n-grams overall (sum TF-IDF across docs, then sort)
terms_ub = tfidf_uni_bi.get_feature_names_out()
weights_sum = np.asarray(X_tfidf_uni_bi.sum(axis=0)).ravel()
order = weights_sum.argsort()[::-1]

print('High-weight n-grams overall:')
for i in order[:25]:
    print(f"{terms_ub[i]:<30} {weights_sum[i]:.2f}")
```

```
TF-IDF (1,2) shape: (1139, 30000)
High-weight n-grams overall:
said                           59.68
mr                             51.81
trump                          49.93
clinton                        31.84
mr trump                       22.36
people                         19.61
percent                        19.10
state                          18.39
new                            16.31
mateen                         16.06
obama                          15.46
campaign                       15.09
police                         14.51
orlando                        13.80
sanders                        13.69
party                          13.21
told                           13.10
president                      13.02
like                           12.62
britain                        12.44
republican                     12.39
states                         12.26
year                           12.02
mrs                            11.76
gun                            11.73
```

2.3 3) Quick comparison: BoW(1,2) vs TF-IDF(1,2)

Train two logis:tic-regression models on the same split: one with BoW(1,2) counts, and one with TF-IDF(1,2). Then compare accuracy.

- Expect BoW to do well on style/source; TF-IDF can be similar or slightly lower because it downweights frequent stylistic tokens.

```
[ ]: # BoW
bow = CountVectorizer(lowercase=True, ngram_range=(1,2), min_df=2,␣
 ↪max_features=30000)
Xtr_bow = bow.fit_transform(X_train_txt); Xte_bow = bow.transform(X_test_txt)
acc_bow = accuracy_score(y_test, LogisticRegression(max_iter=1000).fit(Xtr_bow,␣
 ↪y_train).predict(Xte_bow))

# TF-IDF
tfidf = TfidfVectorizer(lowercase=True, ngram_range=(1,2), min_df=2,␣
 ↪max_features=30000)
Xtr_tf = tfidf.fit_transform(X_train_txt); Xte_tf = tfidf.transform(X_test_txt)
acc_tf = accuracy_score(y_test, LogisticRegression(max_iter=1000).fit(Xtr_tf,␣
 ↪y_train).predict(Xte_tf))

print(f"Accuracy - BoW(1,2):   {acc_bow:.3f}")
print(f"Accuracy - TF-IDF(1,2): {acc_tf:.3f}")
```

```
Accuracy - BoW(1,2):   0.873
Accuracy - TF-IDF(1,2): 0.877
```

2.4 4) Class-wise top TF-IDF terms (diagnostics)

Summing TF-IDF vectors over all docs of a class highlights class-salient n-grams (bigrams often reveal style). This helps you interpret what the model finds distinctive.

```
[ ]: # === Per-class top TF-IDF n-grams ===
terms_cls = tfidf_uni_bi.get_feature_names_out()
X = X_tfidf_uni_bi

top_labels = df[label_col].value_counts().head(3).index.tolist()
for lab in top_labels:
    rows = df[df[label_col] == lab].index
    vec = X[rows].sum(axis=0).A1      # sum TF-IDF across rows of this class
    top_idx = vec.argsort()[::-1][:20]
    print(f"\nTop TF-IDF terms for label: {lab}")
    for j in top_idx:
        print(f"  {terms_cls[j]:<30} {vec[j]:.2f}")
```

```
Top TF-IDF terms for label: New York Times
  mr                             50.94
  said                           27.85
  trump                          23.43
  mr trump                       22.10
  clinton                        11.69
  mrs clinton                    11.47
  mrs                            11.42
  ms                             11.08
```

```
  people                         9.50
  _____                          8.40
  like                           8.17
  new                            7.41
  campaign                       7.19
  state                          6.86
  states                         6.84
  party                          6.61
  united                         6.40
  years                          6.30
  just                           6.25
  britain                        6.17

Top TF-IDF terms for label: Fox News
  trump                          17.63
  clinton                        15.99
  said                           14.63
  mateen                         10.57
  told                           7.31
  sanders                        6.94
  orlando                        6.76
  state                          6.26
  latest                         5.74
  obama                          5.57
  attack                         5.48
  people                         5.45
  isis                           5.42
  police                         5.22
  campaign                       5.12
  department                     4.88
  fbi                            4.80
  democratic                     4.63
  terror                         4.51
  percent                        4.34

Top TF-IDF terms for label: Reuters
  said                           17.19
  percent                        10.42
  trump                          8.87
  state                          5.28
  britain                        5.20
  billion                        5.14
  new                            5.05
  eu                             5.04
  company                        4.78
  vote                           4.67
  people                         4.65
  clinton                        4.16
```

```
friday              4.02
islamic             3.99
market              3.97
year                3.94
obama               3.89
million             3.84
week                3.81
investors           3.78
```

2.5 5) Cosine similarity mini-demo

TF-IDF vectors let us measure document similarity with cosine (angle).

- Values near 1.0 → very similar; closer to 0 → dissimilar.

[ ]:
```python
# === Cosine similarity between documents in TF-IDF space ===
from sklearn.metrics.pairwise import cosine_similarity

A = X_tfidf_uni_bi[df.index[:1]]      # first doc
B = X_tfidf_uni_bi[df.index[1:3]]     # next two docs
cos = cosine_similarity(A, B)
print('Cosine(doc0, doc1) =', round(float(cos[0,0]), 3))
print('Cosine(doc0, doc2) =', round(float(cos[0,1]), 3))
```

```
Cosine(doc0, doc1) = 0.011
Cosine(doc0, doc2) = 0.013
```

- IDF intuition: If a word appears in every article, its IDF is small → it won't dominate.
- When TF-IDF beats BoW: topic retrieval, deduplication, and tasks where frequent function words are noise rather than signal.
- When BoW can win: source/style labeling where frequent collocations are genuinely informative.

---

3 Part 3: Embeddings (CBOW) → Document Vectors

3.1 1) Train Word2Vec (CBOW)

We learn word embeddings from our corpus using Word2Vec (CBOW). CBOW predicts a word from its surrounding context → smooths meaning over local windows.

Parameters to note:

- `vector_size=100` → dimensionality of each word vector (trade-off: expressiveness vs speed).
- `window=5` → how many words left/right define "context".
- `min_count=2` → ignore very rare words (reduces noise, speeds training).
- `sg=0` → CBOW (set sg=1 for Skip-gram, which can help rarer words).
- `epochs=10` → passes over the data (more = better but slower).
- `seed=42` → reproducibility.

Embeddings capture semantic similarity (e.g., price near market), not just shared spelling. This helps when wording changes but meaning doesn't.

[ ]:
```
# === Train Word2Vec (CBOW) on our cleaned corpus ===
try:
    from gensim.models import Word2Vec
except Exception as e:
    print("If gensim is missing, run: pip install gensim")
    raise

# Simple tokenizer consistent with earlier sections
TOKEN_RE = re.compile(r"[A-Za-z']+")
def tokenize(text: str):
    return TOKEN_RE.findall(str(text).lower())

# Tokenize every document; feed to Word2Vec (CBOW)
sentences = [tokenize(t) for t in df['text_norm']]

w2v = Word2Vec(
    sentences=sentences,
    vector_size=100,    # embedding size
    window=5,           # context window
    min_count=2,        # ignore words seen < 2 times
    workers=4,          # CPU threads
    sg=0,               # 0 = CBOW, 1 = Skip-gram
    epochs=10,
    seed=42
)

wv = w2v.wv  # keyed vectors (lookups & similarity live here)
print('Vocab size:', len(wv))
print('Vector size:', wv.vector_size)

# Quick sanity check: nearest neighbors for a few terms (if present)
query_terms = 
 ↪['government','market','court','football','climate','china','inflation','white','house']
for q in query_terms:
    if q in wv.key_to_index:
        print(f"\nMost similar to '{q}':")
        for w, s in wv.most_similar(q, topn=5):
            print(f"  {w:<15} {s:.3f}")
```

```
Vocab size: 20139
Vector size: 100

Most similar to 'government':
  military        0.720
  diplomatic      0.712
```

```
  iran           0.709
  rebels         0.706
  syrian         0.690

Most similar to 'market':
  price          0.833
  markets        0.833
  growth         0.816
  global         0.809
  inflation      0.807

Most similar to 'court':
  ruling         0.815
  supreme        0.798
  decision       0.754
  appeals        0.726
  case           0.713

Most similar to 'football':
  singer         0.795
  lonnie         0.793
  kundla         0.792
  bushmaster     0.774
  russell        0.773

Most similar to 'climate':
  nondisclosure  0.695
  treaty         0.675
  repatriation   0.669
  excess         0.666
  regulatory     0.665

Most similar to 'china':
  korea          0.837
  india          0.810
  iran           0.810
  asia           0.802
  sea            0.799

Most similar to 'inflation':
  growth         0.894
  negative       0.866
  increases      0.860
  oil            0.857
  fuel           0.841

Most similar to 'white':
  representatives 0.654
```

```
  lords           0.610
  thunderbirds    0.599
  speaker         0.586
  waffle          0.541

Most similar to 'house':
  senate          0.712
  cloth           0.672
  bronco          0.660
  supremacists    0.620
  representatives 0.593
```

3.2 2) Build document vectors (mean and TF-IDF-weighted mean)

link text We turn each document into one fixed-length vector by pooling its word vectors. * Mean pooling: average all word vectors in the doc → simple, fast baseline. * TF-IDF-weighted mean: weigh each word vector by its importance in that doc (TF-IDF) → downweights boilerplate.

Edge cases handled:

- If a document has no in-vocab words, we return a zero vector.
- TF-IDF weights are fit on the train split only; we use the trained vocabulary to weight test docs (prevents leakage).

[ ]:
```
# === Turn text into document vectors (two ways) - using the canonical split ===

# Dimension of each word vector from your trained Word2Vec model
EMB_DIM = wv.vector_size

def docvec_mean(tokens, wv, dim):
    """Return the mean (average) of in-vocabulary word vectors.
    If a doc has no tokens in the embedding vocab, return a zero vector."""
    vecs = [wv[t] for t in tokens if t in wv.key_to_index]
    return np.mean(vecs, axis=0) if vecs else np.zeros(dim, dtype=np.float32)

# Tokenize the already-split train/test texts
# (tokenize should match how you trained Word2Vec: lowercase, A-Z + apostrophes)
train_tokens = [tokenize(t) for t in X_train_txt]
test_tokens  = [tokenize(t) for t in X_test_txt]

# --- Fit TF-IDF (on TRAIN only) to get per-term weights for weighting word␣
↪vectors ---
# Using unigrams; you can keep this aligned with your earlier word-model policy␣
↪by
# adding stop_words=STOPWORDS and a token_pattern if desired.
tfidf_train = TfidfVectorizer(
    lowercase=True,
    ngram_range=(1,1),
    min_df=2,
```

```
    max_features=20000
    # Optional for consistency:
    # stop_words=STOPWORDS,
    # token_pattern=r"(?u)\b[a-zA-Z][a-zA-Z'-]{1,}\b"
)

# Learn TF (and IDF) statistics from TRAIN, then transform TRAIN and TEST
Xtr_tfidf = tfidf_train.fit_transform(X_train_txt)
Xte_tfidf = tfidf_train.transform(X_test_txt)

# Map: term -> column index in the TF-IDF matrix (needed to match tokens to␣
↪weights)
term_to_col = tfidf_train.vocabulary_

def docvec_tfidf(tokens, tfidf_row, wv, dim, term_to_col):
    """Return a TF-IDF-weighted average of word vectors for one doc.
    If all weights are zero or no tokens are in-vocab, fall back to mean/zeros.
↪"""
    if tfidf_row.nnz == 0:  # quick exit if the row is empty
        return np.zeros(dim, dtype=np.float32)

    # Sparse row -> dict of {column_index: weight} for fast lookup
    inds = tfidf_row.indices
    data = tfidf_row.data
    col_to_w = {col: wt for col, wt in zip(inds, data)}

    acc = np.zeros(dim, dtype=np.float32)
    Z = 0.0  # normalization constant (sum of weights actually used)
    for t in tokens:
        col = term_to_col.get(t)             # which column corresponds to␣
↪token t?
        if col is not None and t in wv.key_to_index:
            w = col_to_w.get(col, 0.0)       # TF-IDF weight for token t in␣
↪THIS doc
            if w > 0:
                acc += w * wv[t]             # accumulate weighted word vector
                Z += w
    return acc / Z if Z > 0 else docvec_mean(tokens, wv, dim)

# --- Build the two document-vector matrices (TRAIN and TEST) ---

# 1) Mean-pooled embeddings
Xtr_emb_mean = np.vstack([docvec_mean(tok, wv, EMB_DIM) for tok in␣
↪train_tokens])
Xte_emb_mean = np.vstack([docvec_mean(tok, wv, EMB_DIM) for tok in test_tokens])

# 2) TF-IDF-weighted mean embeddings
```

```
#    (use the TF-IDF row for each document to weight its tokens)
Xtr_emb_tfidf = np.vstack([
    docvec_tfidf(train_tokens[i], Xtr_tfidf[i], wv, EMB_DIM, term_to_col)
    for i in range(len(train_tokens))
])
Xte_emb_tfidf = np.vstack([
    docvec_tfidf(test_tokens[i],  Xte_tfidf[i], wv, EMB_DIM, term_to_col)
    for i in range(len(test_tokens))
])

# Shapes sanity check: (n_train, emb_dim), (n_test, emb_dim) for both variants
Xtr_emb_mean.shape, Xte_emb_mean.shape, Xtr_emb_tfidf.shape, Xte_emb_tfidf.shape
```

[ ]: ((911, 100), (228, 100), (911, 100), (228, 100))

3.3 3) Baseline classifier on document embeddings

Train Logistic Regression on the two document-vector variants and compare accuracy & per-class metrics.

- Expect these to trail strong n-gram baselines for source/style prediction.
- They shine more on semantic tasks (see the later search demo).

[ ]:
```
# === Classify with doc embeddings ===
clf_mean = LogisticRegression(max_iter=2000).fit(Xtr_emb_mean, y_train)
pred_mean = clf_mean.predict(Xte_emb_mean)

clf_w = LogisticRegression(max_iter=2000).fit(Xtr_emb_tfidf, y_train)
pred_w = clf_w.predict(Xte_emb_tfidf)

print('Accuracy - Mean embeddings :', round(accuracy_score(y_test, pred_mean),␣
 ↪3))
print('Accuracy - TF-IDF-weighted:', round(accuracy_score(y_test, pred_w), 3))
print('\nPer-class report (weighted embeddings):')
print(classification_report(y_test, pred_w))
```

```
Accuracy - Mean embeddings : 0.794
Accuracy - TF-IDF-weighted: 0.759

Per-class report (weighted embeddings):
                precision    recall  f1-score   support

      Fox News       0.65      0.69      0.67        61
New York Times       0.92      0.79      0.85       110
       Reuters       0.65      0.77      0.70        57

      accuracy                           0.76       228
     macro avg       0.74      0.75      0.74       228
```

```
weighted avg       0.78      0.76      0.76       228
```

3.4 4) Visualize a slice of the word space

Runs PCA to 2D on a small subset of word vectors, scatter them, and label a few points. It's only for intuition—clusters suggest semantic neighborhoods.

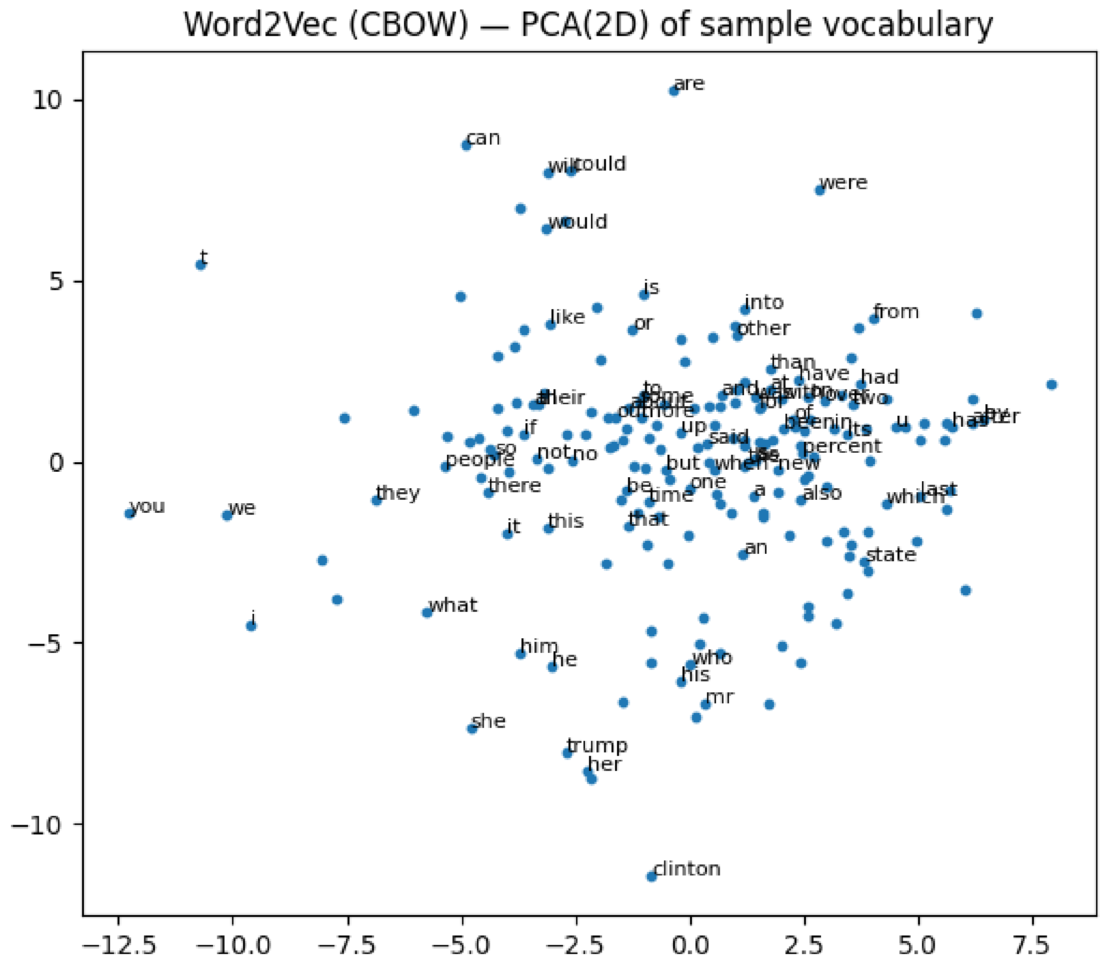
- CBOW vs Skip-gram: CBOW is faster and smooths frequent words; Skip-gram can model rarer words better.
- Where embeddings shine: semantic similarity / retrieval, clustering, transfer to other tasks.

[ ]:
```
# === Visualize a tiny slice of the learned word space (optional) ===
from sklearn.decomposition import PCA

vocab_items = list(wv.key_to_index.keys())[:200]            # small sample so␣
↪it's readable
vecs = np.vstack([wv[w] for w in vocab_items])
xy = PCA(n_components=2, random_state=42).fit_transform(vecs)

plt.figure(figsize=(7,6))
plt.scatter(xy[:,0], xy[:,1], s=10)
for (x, y), w in zip(xy, vocab_items[:80]):                 # label a subset
    plt.text(x, y, w, fontsize=8)

plt.title('Word2Vec (CBOW) - PCA(2D) of sample vocabulary')
plt.show()
```



3.5 5) Unified mini-benchmark: BoW / TF-IDF / Char n-grams / Local CBOW / Pretrained W2V (GoogleNews)

Runs a side-by-side comparison of 12 lightweight text classifiers on the same train/test split. You'll see how simple count features, TF-IDF weighting, character n-grams, and CBOW document embeddings stack up for outlet classification. We keep tokenization/stopword policy consistent across the word models for a fair fight, reuse the already-trained Word2Vec (wv) for embeddings, and report accuracy and macro-F1.

Love this section—it's a super clean sandbox for showing how small choices change behavior. Here's prose you can paste right under the code. It explains **what each method does**, **why it differs**, and **when it tends to shine**.

---

4 What the 8 methods are doing

All eight models share the same basic recipe: **(a)** turn text into a big sparse feature vector, **(b)** train a fast linear classifier. They differ along four knobs:

- **Tokenizer/units:** word unigrams–bigrams vs **character** 3–5-grams
- **Weighting:** raw counts, binary (0/1), TF (length-normalized counts), or **TF-IDF**
- **Scaling:** L2 normalization or not; optional **sublinear TF** (log-scaling within a doc)
- **Classifier:** Logistic Regression (LR) vs **LinearSVC** (linear SVM)

Below, each method in plain English.

4.0.1 1) BoW(1,2) counts + Logistic Regression

- **What:** Word 1-grams and 2-grams; features are raw counts.
- **Why it's different:** No global down-weighting of common words; the model must learn to ignore frequent tokens on its own.
- **When it works:** Source/style labeling, short texts, or when bigrams carry strong cues ("white house", "golden state").
- **Trade-offs:** Long documents get larger feature magnitudes, which can dominate unless regularization reins them in.

4.0.2 2) BoW(1,2) binary + Logistic Regression

- **What:** Same n-grams, but each feature is 0/1 ("did this token appear?").
- **Why it's different:** Neutralizes document length and repeated words; focuses on **presence** not frequency.
- **When it works:** Style/outlet tasks where **having** a phrase matters more than how often it appears; heterogeneous article lengths.
- **Trade-offs:** You throw away within-doc frequency signal that sometimes helps.

4.0.3 3) TF only (L2-normalized) + Logistic Regression

- **What:** Term Frequency (no IDF), then L2 row-normalization to unit length.
- **Why it's different:** Like counts, but explicitly controls for length. A middle ground between (1) and TF-IDF.
- **When it works:** Mixed-length corpora where repetition shouldn't overrule diversity of terms.
- **Trade-offs:** Still treats globally common words as informative unless the classifier down-weights them.

4.0.4 4) TF-IDF(1,2) (defaults) + Logistic Regression

- **What:** Classic TF-IDF with L2 norm.

- **Why it's different:** Down-weights tokens that appear in many documents; promotes rare, discriminative n-grams.
- **When it works:** Most general text classification; often the **strongest lexical baseline**.
- **Trade-offs:** If your label truly correlates with very common words (rare), IDF can over-down-weight them.

4.0.5 5) TF-IDF(1,2) with sublinear TF + Logistic Regression

- **What:** Same as (4) but apply **log(1+tf)** within each document before IDF and normalization.
- **Why it's different:** Reduces the influence of spammy repetition ("buy buy buy"), keeping "once vs ten times" from dwarfing other signals.
- **When it works:** Headlines + bodies, social posts, or any domain with bursty word repeats.
- **Trade-offs:** If frequency genuinely matters (e.g., intensity markers), log-scaling can mute useful signal.

4.0.6 6) TF-IDF sublinear, no norm + Logistic Regression

- **What:** Same features as (5) **but no L2 normalization**.
- **Why it's different:** Preserves overall magnitude—so longer, denser documents contribute **larger** vectors.
- **When it works:** If document length or overall "content mass" correlates with the label.
- **Trade-offs:** Can hurt fairness across lengths; be cautious on varied-length corpora.

4.0.7 7) TF-IDF (character 3–5) + Logistic Regression

- **What:** Character n-grams (3–5) rather than word tokens.
- **Why it's different:** Captures **morphology, punctuation, casing, name shapes, typos**—great for style and robustness (handles OOV words, misspellings, hashtags).
- **When it works:** Author profiling, outlet/style ID, noisy text (social media), multilingual/Code-Switching.
- **Trade-offs:** Less semantic: it won't "understand" topics as deeply as word n-grams; feature spaces can get big.

4.0.8 8) TF-IDF(1,2) + LinearSVC

- **What:** Same lexical features as (4), but classifier is a **linear SVM** (hinge loss) instead of LR (log loss).
- **Why it's different:** SVM focuses on **maximizing the margin**; LR models calibrated probabilities. In high-dim sparse text, SVM often ties or slightly edges LR.
- **When it works:** Most of the time; especially when classes are separable with wide margins.
- **Trade-offs:** No native probabilities (need Platt scaling if you want them). Sensitivity to C can differ from LR.

4.1 Quick cheat sheet (differences at a glance)

- **Counts vs Binary vs TF vs TF-IDF:**
  - Counts = raw frequency; Binary = presence only; TF = length-controlled counts; **TF-IDF** = TF with global rarity boost.
- **Sublinear TF:** dampens repetition; often a small, consistent win.
- **Normalization (L2):** makes documents comparable regardless of length. Turning it **off** lets length/magnitude matter.
- **Word vs Char n-grams:** words capture semantics and phrases; **chars** capture style, morphology, OOV robustness.
- **LR vs LinearSVC:** LR gives probabilities; SVM pushes a margin and is a perennial strong baseline for sparse TF-IDF.

Some takeaways:

- (4) is your **default strong lexical baseline**.
- (7) wins when **style/orthography** matters or text is noisy.
- (2)/(3) are "length-robust" variants.
- (8) is "same features, different loss"—often a tiny bump over (4).

[ ]:
```python
# === Unified mini-benchmark: BoW / TF-IDF / Char n-grams / Base Embeddings␣
↪(CBOW) ===

# Token pattern: letters + apostrophes/hyphens (keeps "mr/ms", "don't",␣
↪"e-mail"; drops pure numbers)
TOK = r"(?u)\b[a-zA-Z][a-zA-Z'-]{1,}\b"

# ---------- Canonical split (already created earlier) ----------
Xtr, Xte, ytr, yte = X_train_txt, X_test_txt, y_train, y_test  # reuse across␣
↪all models

# ---------- Helper to run and log experiments ----------
experiments = []
def run(name, pipe):
    """Fit a pipeline on the train split, predict on the test split, log␣
↪accuracy + macro-F1."""
    pipe.fit(Xtr, ytr)
    pred = pipe.predict(Xte)
    experiments.append({
        "model": name,
        "acc": accuracy_score(yte, pred),
        "macro_f1": f1_score(yte, pred, average="macro")
```

```
    })

# ---------- 1) BoW (counts, uni+bi) + Logistic Regression ----------
# Classic counts; good baseline for style/source tasks.
run("BoW(1,2) counts + LR",
    make_pipeline(
        CountVectorizer(ngram_range=(1,2), min_df=2, max_features=30000,
                        stop_words=STOPWORDS, token_pattern=TOK),
        LogisticRegression(max_iter=1500)
    )
)

# ---------- 2) BoW (binary presence, uni+bi) + LR ----------
# Presence/absence neuters doc length; often best for outlet/style signals.
run("BoW(1,2) binary + LR",
    make_pipeline(
        CountVectorizer(ngram_range=(1,2), min_df=2, max_features=30000,␣
 ↪binary=True,
                        stop_words=STOPWORDS, token_pattern=TOK),
        LogisticRegression(max_iter=1500)
    )
)

# ---------- 3) TF-only (L2-normalized) + LR ----------
# TF (no IDF) but with L2 normalization; like counts with length control.
run("TF only (L2) + LR",
    make_pipeline(
        CountVectorizer(ngram_range=(1,2), min_df=2, max_features=30000,
                        stop_words=STOPWORDS, token_pattern=TOK),
        TfidfTransformer(use_idf=False, norm='l2'),
        LogisticRegression(max_iter=1500)
    )
)

# ---------- 4) TF-IDF (uni+bi, defaults) + LR ----------
# Standard TF-IDF; downweights very common tokens globally.
run("TF-IDF(1,2) default + LR",
    make_pipeline(
        TfidfVectorizer(ngram_range=(1,2), min_df=2, max_features=30000,
                        stop_words=STOPWORDS, token_pattern=TOK),
        LogisticRegression(max_iter=1500)
    )
)

# ---------- 5) TF-IDF (sublinear TF) + LR ----------
# Sublinear TF (log scaling) reduces the impact of huge raw counts.
run("TF-IDF(1,2) sublinear_tf + LR",
```

```
    make_pipeline(
        TfidfVectorizer(ngram_range=(1,2), min_df=2, max_features=30000,␣
↪sublinear_tf=True,
                        stop_words=STOPWORDS, token_pattern=TOK),
        LogisticRegression(max_iter=1500)
    )
)

# ---------- 6) TF-IDF (sublinear TF, no norm) + LR ----------
# Keeps magnitude (more BoW-like) while still log-scaling counts.
run("TF-IDF(1,2) sublinear, no norm + LR",
    make_pipeline(
        TfidfVectorizer(ngram_range=(1,2), min_df=2, max_features=30000,
                        sublinear_tf=True, norm=None,
                        stop_words=STOPWORDS, token_pattern=TOK),
        LogisticRegression(max_iter=1500)
    )
)

# ---------- 7) TF-IDF (character 3-5) + LR ----------
# Char n-grams capture surface/style (affixes, punctuation, name shapes).
# Note: stop_words/token_pattern are ignored for char analyzers.
run("TF-IDF char 3-5 + LR",
    make_pipeline(
        TfidfVectorizer(analyzer='char', ngram_range=(3,5), min_df=5,␣
↪sublinear_tf=True),
        LogisticRegression(max_iter=1500)
    )
)

# ---------- 8) TF-IDF (uni+bi) + LinearSVC ----------
# Same features as (4), different linear classifier (often ties/edges LR).
run("TF-IDF(1,2) default + LinearSVC",
    make_pipeline(
        TfidfVectorizer(ngram_range=(1,2), min_df=2, max_features=30000,
                        stop_words=STOPWORDS, token_pattern=TOK),
        LinearSVC()
    )
)
```

[ ]:
```
# ---------- 9-10) BASE EMBEDDINGS (CBOW) - reuse the already-trained `wv` from␣
↪earlier ----------
try:
    wv  # from Part 3
    D = wv.vector_size
except NameError:
```

```
    print("Embeddings rows skipped → no `wv` found. Run the Part 3 Word2Vec␣
↪training cell first.")
else:
    # Use existing tokenize(); if missing, define a compatible fallback␣
↪matching W2V training.
    if 'tokenize' not in globals():
        TOKEN_RE = re.compile(r"[A-Za-z']+")
        def tokenize(text: str):
            return TOKEN_RE.findall(str(text).lower())

    def docvec_mean(tokens, wv, dim):
        """
        Build a document vector by simple mean pooling:
        - keep only tokens that exist in the W2V vocabulary
        - average their vectors
        - if nothing is in-vocab, return a zero vector of the right size to␣
↪avoid NaNs.
        """
        vecs = [wv[t] for t in tokens if t in wv.key_to_index]
        return np.mean(vecs, axis=0) if vecs else np.zeros(dim, dtype=np.
↪float32)

    # Tokenize the train/test splits once
    Xtr_tok = [tokenize(t) for t in Xtr]
    Xte_tok = [tokenize(t) for t in Xte]

    # 9) Mean-pooled embeddings + LR. Mean pooling treats every in-vocab word␣
↪equally.
    # Convert each doc into a mean-pooled embedding, then train a linear␣
↪classifier on top.
    Xtr_emb_mean = np.vstack([docvec_mean(tok, wv, D) for tok in Xtr_tok])
    Xte_emb_mean = np.vstack([docvec_mean(tok, wv, D) for tok in Xte_tok])
    clf = LogisticRegression(max_iter=2000).fit(Xtr_emb_mean, ytr)
    pred = clf.predict(Xte_emb_mean)
    experiments.append({
        "model": "CBOW mean emb + LR",
        "acc": accuracy_score(yte, pred),
        "macro_f1": f1_score(yte, pred, average="macro")
    })

    # 10) TF-IDF-weighted mean embeddings (fit weights on TRAIN only)
    # TF-IDF pooling gives rare/diagnostic words more pull than common ones.
    tf_uni = TfidfVectorizer(
        lowercase=True, ngram_range=(1,1), min_df=2, max_features=20000,
        stop_words=STOPWORDS, token_pattern=TOK
    )
```

```python
Xtr_tf = tf_uni.fit_transform(Xtr)   # learn weights on TRAIN
Xte_tf = tf_uni.transform(Xte)       # apply to TEST
term_to_col = tf_uni.vocabulary_

def docvec_tfidf(tokens, tf_row, wv, dim, term_to_col):
    """
    TF-IDF-weighted pooling:
    - Look up each token's TF-IDF weight for THIS document (tf_row).
    - Accumulate weight * word_vector, and divide by total weight Z.
    - If the row is empty or no tokens overlap, fall back to mean/zeros for stability.
    """
    if tf_row.nnz == 0:
        return np.zeros(dim, dtype=np.float32)
    inds = tf_row.indices; data = tf_row.data           # sparse row -> indices & weights
    col2w = {c:w for c, w in zip(inds, data)}           # column index -> TF-IDF weight
    acc = np.zeros(dim, dtype=np.float32); Z = 0.0
    for t in tokens:
        col = term_to_col.get(t)                        # where this token lives in the TF-IDF row
        if col is not None and t in wv.key_to_index:    # only pool tokens in both TF-IDF and W2V vocab
            w = col2w.get(col, 0.0)
            if w > 0:
                acc += w * wv[t]; Z += w
    return acc / Z if Z > 0 else docvec_mean(tokens, wv, dim)

# Build weighted doc vectors for train/test using the TRAIN-fitted TF-IDF weights.
Xtr_emb_w = np.vstack([docvec_tfidf(Xtr_tok[i], Xtr_tf[i], wv, D, term_to_col) for i in range(len(Xtr_tok))])
Xte_emb_w = np.vstack([docvec_tfidf(Xte_tok[i], Xte_tf[i], wv, D, term_to_col) for i in range(len(Xte_tok))])

# Same linear head, now over TF-IDF-weighted embeddings.
clf = LogisticRegression(max_iter=2000).fit(Xtr_emb_w, ytr)
pred = clf.predict(Xte_emb_w)
experiments.append({
    "model": "CBOW TF-IDF-weighted emb + LR",
    "acc": accuracy_score(yte, pred),
    "macro_f1": f1_score(yte, pred, average="macro")
})
```

[ ]:
```python
# ---------- 11-12) PRETRAINED WORD2VEC (GoogleNews) ----------
# Compares a strong off-the-shelf embedding baseline against the␣
↪locally-trained CBOW.

# 11a) Load GoogleNews (300-d). If already cached/loaded earlier, this is␣
↪instant.
kv = api.load("word2vec-google-news-300")
Dk = kv.vector_size  # 300

# 11b) Use the same token rules as your other vectorizers so comparisons are␣
↪fair
# (same casing, stopwords, token pattern).
analyzer_kv = CountVectorizer(
    lowercase=True, token_pattern=TOK, stop_words=STOPWORDS
).build_analyzer()

def kv_vec(tok: str):
    """
    Look up a token in the GoogleNews keyed vectors.
    GoogleNews is case-sensitive and contains many capitalized forms, so
    we try the exact token first, then a simple Capitalize() fallback.
    """
    if tok in kv.key_to_index:
        return kv[tok]
    tcap = tok.capitalize()
    if tcap in kv.key_to_index:
        return kv[tcap]
    return None  # out-of-vocab → ignored in pooling

def docvec_mean_kv(tokens):
    """
    Mean-pool the pretrained word vectors that exist for this document.
    If nothing is in-vocab, return a zero vector (prevents NaNs).
    """
    vs = []
    for t in tokens:
        v = kv_vec(t)
        if v is not None:
            vs.append(v)
    return np.mean(vs, axis=0) if vs else np.zeros(Dk, dtype=np.float32)

# Tokenize the canonical split (same split you used everywhere else)
Xtr_tok_kv = [analyzer_kv(t) for t in Xtr]
Xte_tok_kv = [analyzer_kv(t) for t in Xte]

# 11) Pretrained W2V mean-pooled + LR
# Build doc embeddings by simple mean pooling, then train a linear head.
```

```
Xtr_kv_mean = np.vstack([docvec_mean_kv(tok) for tok in Xtr_tok_kv])
Xte_kv_mean = np.vstack([docvec_mean_kv(tok) for tok in Xte_tok_kv])

clf = LogisticRegression(max_iter=2000).fit(Xtr_kv_mean, ytr)
pred = clf.predict(Xte_kv_mean)
experiments.append({
    "model": "W2V GoogleNews mean emb + LR",
    "acc": accuracy_score(yte, pred),
    "macro_f1": f1_score(yte, pred, average="macro")
})

# 12) Pretrained W2V TF-IDF-weighted + LR
# Reuse (10)'s 1-gram TF-IDF if it exists to avoid refitting; else fit on TRAIN␣
↪only (no leakage).
if "tf_uni" in globals() and "Xtr_tf" in globals() and "Xte_tf" in globals():
    tf_for_emb = tf_uni
    Xtr_tf_for_emb = Xtr_tf
    Xte_tf_for_emb = Xte_tf
else:
    tf_for_emb = TfidfVectorizer(
        lowercase=True, ngram_range=(1,1), min_df=2, max_features=20000,
        stop_words=STOPWORDS, token_pattern=TOK
    )
    Xtr_tf_for_emb = tf_for_emb.fit_transform(Xtr)   # learn weights on TRAIN␣
↪only
    Xte_tf_for_emb = tf_for_emb.transform(Xte)       # apply same mapping to␣
↪TEST

# Map token → column index once, and keep CSR for fast per-row lookups.
term_to_col = tf_for_emb.vocabulary_
Xtr_tf_for_emb = Xtr_tf_for_emb.tocsr()
Xte_tf_for_emb = Xte_tf_for_emb.tocsr()

def docvec_tfidf_kv(tokens, tf_row):
    """
    TF-IDF-weighted pooling with pretrained vectors:
      - For this document's sparse TF-IDF row, build {column → weight}.
      - For each token that appears in BOTH the TF-IDF vocab and kv vocab,
        accumulate weight * vector.
      - Normalize by total weight Z; if Z==0, fall back to mean pooling.
    """
    if tf_row.nnz == 0:
        return np.zeros(Dk, dtype=np.float32)
    col2w = {c: w for c, w in zip(tf_row.indices, tf_row.data)}
    acc = np.zeros(Dk, dtype=np.float32); Z = 0.0
    for t in tokens:
        col = term_to_col.get(t)
```

```
        if col is not None:
            v = kv_vec(t)           # may be None if OOV in GoogleNews
            if v is not None:
                w = col2w.get(col, 0.0)
                if w > 0:
                    acc += w * v; Z += w
    return acc / Z if Z > 0 else docvec_mean_kv(tokens)

# Build weighted doc vectors for train/test using TRAIN-fitted TF-IDF weights.
Xtr_kv_w = np.vstack([docvec_tfidf_kv(Xtr_tok_kv[i], Xtr_tf_for_emb[i]) for i␣
↪in range(len(Xtr_tok_kv))])
Xte_kv_w = np.vstack([docvec_tfidf_kv(Xte_tok_kv[i], Xte_tf_for_emb[i]) for i␣
↪in range(len(Xte_tok_kv))])

# Same linear head, now over TF-IDF-weighted pretrained embeddings.
clf = LogisticRegression(max_iter=2000).fit(Xtr_kv_w, ytr)
pred = clf.predict(Xte_kv_w)
experiments.append({
    "model": "W2V GoogleNews TF-IDF-weighted emb + LR",
    "acc": accuracy_score(yte, pred),
    "macro_f1": f1_score(yte, pred, average="macro")
})

# ---------- Results table ----------
# Round for readability; sort by macro-F1 then accuracy (macro-F1 is safer on␣
↪class imbalance).
res = pd.DataFrame(experiments)
res['acc'] = res['acc'].round(3)
res['macro_f1'] = res['macro_f1'].round(3)
print(res.sort_values(["macro_f1","acc"], ascending=False).
↪to_string(index=False))
```

```
[==================================================] 100.0% 1662.8/1662.8MB
downloaded
                                  model   acc  macro_f1
    TF-IDF(1,2) sublinear, no norm + LR 0.877     0.860
                   BoW(1,2) binary + LR 0.873     0.858
                   TF-IDF char 3-5 + LR 0.868     0.846
             TF-IDF(1,2) sublinear_tf + LR 0.868     0.845
           TF-IDF(1,2) default + LinearSVC 0.851     0.838
                TF-IDF(1,2) default + LR 0.838     0.817
                      TF only (L2) + LR 0.829     0.813
                   BoW(1,2) counts + LR 0.811     0.796
                     CBOW mean emb + LR 0.794     0.775
          CBOW TF-IDF-weighted emb + LR 0.728     0.701
           W2V GoogleNews mean emb + LR 0.724     0.696
W2V GoogleNews TF-IDF-weighted emb + LR 0.689     0.664
```

4.2 6) Error Analysis — Confusion Matrix & Top Features (Just locally trained Word2Vec CBOW)

Fits (or reuses) a strong binary BoW (1,2) baseline, then:

1. prints a confusion matrix to show which outlets get confused with which, and
2. lists the top weighted features per class from the logistic regression to interpret what the model is actually using.

Confusions highlight boundary cases; top features make the model's behavior auditable (and help spot residual leakage).

[ ]:
```
bin_bow = make_pipeline(
    CountVectorizer(
        ngram_range=(1,2),       # unigrams + bigrams capture short collocations
        min_df=2,                # drop ultra-rare terms
        max_features=30000,      # cap vocab size
        binary=True              # presence/absence (neutralizes doc length)
        # Optional (recommended for consistency with earlier cells):
        # , stop_words=STOPWORDS, token_pattern=r"(?
 ↪u)\b[a-zA-Z][a-zA-Z'-]{1,}\b"
    ),
    LogisticRegression(max_iter=1500)  # fast, strong linear baseline on sparse␣
 ↪text
)

# Fit on the canonical split and predict
bin_bow.fit(Xtr, ytr)
pred = bin_bow.predict(Xte)

# --- Confusion matrix (rows = true labels, cols = predicted labels) ---
cm = confusion_matrix(yte, pred)
cm_df = pd.DataFrame(
    cm,
    index=bin_bow.named_steps['logisticregression'].classes_,
    columns=bin_bow.named_steps['logisticregression'].classes_
)
cm_df  # display nicely in the notebook

# --- Top features per class (largest positive weights in the one-vs-rest␣
 ↪logits) ---
vec = bin_bow.named_steps['countvectorizer']
clf = bin_bow.named_steps['logisticregression']
terms = vec.get_feature_names_out()

for cls, coef in zip(clf.classes_, clf.coef_):
    # coef is the weight vector for "this class vs rest"
    top = np.argsort(coef)[-15:][::-1]  # indices of the 15 largest positive␣
 ↪weights
```

```python
    print(f"\nTop features for {cls}:")
    for j in top:
        print(f"  {terms[j]:<25} {coef[j]:.3f}")
```

```
Top features for Fox News:
  told                      0.228
  from the                  0.183
  says                      0.179
  terror                    0.163
  com                       0.160
  2016                      0.160
  during                    0.160
  told com                  0.155
  gop                       0.155
  elections from            0.154
  politics see              0.154
  see latest                0.154
  2016 elections            0.154
  biggest name              0.154
  latest coverage           0.154

Top features for New York Times:
  mr                        0.708
  ms                        0.292
  mr trump                  0.220
  said that                 0.192
  had been                  0.171
  here                      0.171
  that he                   0.166
  that mr                   0.161
  of mr                     0.153
  case                      0.143
  and mr                    0.142
  that would                0.141
  here are                  0.132
  and that                  0.130
  far                       0.128

Top features for Reuters:
  additional                0.325
  its                       0.250
  inc                       0.243
  said on                   0.241
  president barack          0.210
  barack obama              0.197
  barack                    0.194
```

```
on friday                  0.194
said                       0.191
percent                    0.184
week                       0.166
it                         0.157
said it                    0.143
state                      0.143
united                     0.138
```

4.3 7) Semantic Search — TF-IDF (lexical) vs Local CBOW vs Pretrained Word2Vec (semantic)

Builds three search indexes over the corpus: 1. a TF-IDF index that retrieves documents based on shared words and n-grams (lexical overlap), 2. a local CBOW embedding index that retrieves based on semantic similarity learned from this corpus, and 3. a pretrained Word2Vec (GoogleNews) index that retrieves based on semantic similarity learned from large-scale, general English text.

You can call:

`compare_search_v3("your query", k=5)`

to view the top-k results from all systems side-by-side, or limit to certain scorers, e.g.:

`compare_search_v3("hurricane landfall florida panhandle", k=5, show=("tfidf", "gn_mean", "local_mean"))`

This shows how retrieval evolves: * TF-IDF excels when the query shares exact words or phrases with the document. * Local CBOW can bridge minor rewordings using context learned from your dataset. * Pretrained Word2Vec captures broader paraphrases and synonymy, often finding meaning-matched results even with no shared vocabulary.

[ ]:
```python
# Make sure the normalized text column exists before we build any indexes.
assert 'text_norm' in df.columns, "df['text_norm'] missing"

# Figure out which optional columns we can use for pretty-printing results.
# If a column is missing, store None so we can fall back to empty strings later.
TITLES_COL = 'title' if 'title' in df.columns else None
PUBS_COL   = 'publication' if 'publication' in df.columns else None

# --- Convert the dataframe columns into plain Python lists for fast access ---

# Main text to index/search. Replace NaN with "" so downstream tokenizers don't␣
↪crash,
# cast to str for safety (mixed types), then convert to a list.
DOCS = df['text_norm'].fillna("").astype(str).tolist()

# Titles to display alongside hits. If the column is present, clean it like␣
↪above;
# otherwise, create a list of empty strings with the same length as DOCS
# so code like TITLES[i] is always valid.
```

```
TITLES = df[TITLES_COL].fillna("").astype(str).tolist() if TITLES_COL else [""]␣
↪* len(DOCS)

# Publication/source labels for display. Same pattern as TITLES.
PUBS = df[PUBS_COL].fillna("").astype(str).tolist() if PUBS_COL else [""] *␣
↪len(DOCS)

# Pretty-printer for results
def _fmt(i, score):
    pub = f"[{PUBS[i]}] " if PUBS and PUBS[i] else ""
    title = TITLES[i][:100].replace("\n"," ")
    return f"{score:5.3f}  {pub}{title}"
```

[ ]:

```
# === 7 Unified Semantic Search - TF-IDF vs Local CBOW vs Pretrained W2V ===

# 0) Build a dedicated lexical TF-IDF for semantic search
#    Rationale: we "freeze" a private TF-IDF (tfidf_lex7) so the doc index and
#    query vectors ALWAYS share the same vocabulary/columns (prevents dim␣
↪mismatches).
tfidf_lex7 = TfidfVectorizer(
    ngram_range=(1,2),      # use unigrams + bigrams for lexical matching
    min_df=2,               # ignore very rare n-grams
    max_df=0.7,             # ignore extremely common n-grams (stopword-ish)
    sublinear_tf=True,      # log(1 + tf) to damp huge within-doc counts
    stop_words=STOPWORDS,   # same stoplist used elsewhere for fairness
    token_pattern=TOK,      # your "letters + apostrophes/hyphens" token rule
)
X_tfidf_lex7 = tfidf_lex7.fit_transform(DOCS)  # document-term matrix for the␣
↪corpus
analyzer7 = tfidf_lex7.build_analyzer()        # tokenizer/analyzer consistent␣
↪with index
term_to_col7 = tfidf_lex7.vocabulary_          # map: token -> column index␣
↪(used for weighting)
Xtfidf_rows7 = X_tfidf_lex7.tocsr()            # CSR for fast per-row (per-doc)␣
↪lookups
Nlex7 = X_tfidf_lex7.shape[1]                  # number of lexical features in␣
↪the index

def _row_norm(M):
    """
    L2-normalize each row of a dense matrix (so cosine == dot product).
    If a row is all zeros, leave it as zeros (avoid divide-by-zero).
    """
    n = np.linalg.norm(M, axis=1, keepdims=True)
    n[n == 0] = 1.0
    return M / n
```

```python
# 1) Local CBOW (if you trained `wv` earlier)
#    We create two doc-embedding matrices from your locally-trained Word2Vec:
#    - mean-pooled embeddings
#    - TF-IDF-weighted mean embeddings (using the SAME tfidf_lex7 weights)
X_loc_mean7_n = X_loc_tfidf7_n = None
if 'wv' in globals():
    Dloc = wv.vector_size
    # Use the same tokenizer you used to train `wv` if present; otherwise, fall␣
↪back to the TF-IDF analyzer.
    tok_local = tokenize if 'tokenize' in globals() else analyzer7

    def docvec_mean_local7(tokens):
        """
        Simple mean pooling over local W2V:
        - keep only tokens that exist in the W2V vocab
        - average their vectors; fallback to zeros if none are in-vocab
        """
        vecs = [wv[t] for t in tokens if t in wv.key_to_index]
        return np.mean(vecs, axis=0) if vecs else np.zeros(Dloc, dtype=np.
↪float32)

    def docvec_tfidf_local7(tokens, tfidf_row):
        """
        TF-IDF-weighted pooling over local W2V for ONE document:
        - tfidf_row is the sparse vector for that doc from tfidf_lex7
        - for tokens present in BOTH tfidf vocab and W2V vocab:
            accumulate (tfidf_weight * word_vector)
        - normalize by total weight Z; if Z==0, fall back to plain mean pooling
        """
        if tfidf_row.nnz == 0:
            return np.zeros(Dloc, dtype=np.float32)
        col2w = {c: w for c, w in zip(tfidf_row.indices, tfidf_row.data)}  #␣
↪column -> weight
        acc = np.zeros(Dloc, dtype=np.float32); Z = 0.0
        for t in tokens:
            col = term_to_col7.get(t)                   # which column would␣
↪this token map to?
            if col is not None and t in wv.key_to_index:
                w = col2w.get(col, 0.0)                 # TF-IDF weight for THIS␣
↪doc, else 0
                if w > 0:
                    acc += w * wv[t]; Z += w
        return acc / Z if Z > 0 else docvec_mean_local7(tokens)

    # Tokenize every document once (either with your W2V tokenizer or the␣
↪TF-IDF analyzer).
```

```
    TOKS_LOC7 = [tok_local(t) for t in DOCS]

    # Build both embedding variants for all docs
    X_loc_mean7   = np.vstack([docvec_mean_local7(t) for t in TOKS_LOC7])
    X_loc_tfidf7  = np.vstack([docvec_tfidf_local7(TOKS_LOC7[i],␣
↪Xtfidf_rows7[i]) for i in range(len(TOKS_LOC7))])

    # L2-normalize rows so we can use fast cosine via dot products later
    X_loc_mean7_n  = _row_norm(X_loc_mean7)
    X_loc_tfidf7_n = _row_norm(X_loc_tfidf7)
```

[ ]:
```
# 2) Pretrained GoogleNews W2V (300d)
#    Build two document-embedding matrices using *pretrained* Word2Vec:
#    (a) mean-pooled, (b) TF-IDF-weighted mean - both aligned to tfidf_lex7's␣
↪tokens.

kv = api.load("word2vec-google-news-300")  # downloads once, then cached
Dk = kv.vector_size                          # 300-d embeddings

# Tokenize every document with the SAME analyzer as the lexical index.
# This keeps tokenization consistent across TF-IDF and embedding paths.
TOKS_KV7 = [analyzer7(t) for t in DOCS]

def kv_vec7(tok: str):
    """
    Look up a token in GoogleNews vectors.
    GoogleNews is case-sensitive and contains many Capitalized/Title_Case forms,
    so try exact, then Capitalize(), then Title_Case for underscore phrases.
    """
    if tok in kv.key_to_index:
        return kv[tok]
    cap = tok.capitalize()
    if cap in kv.key_to_index:
        return kv[cap]
    if "_" in tok:
        tcap = "_".join(w.capitalize() for w in tok.split("_"))
        if tcap in kv.key_to_index:
            return kv[tcap]
    return None  # OOV → ignore in pooling

def docvec_mean_kv7(tokens):
    """
    Simple mean pooling with pretrained vectors.
    Returns zero vector if no tokens are in-vocab to avoid NaNs.
    """
    vecs = [kv_vec7(t) for t in tokens]
    vecs = [v for v in vecs if v is not None]
```

```
    return np.mean(vecs, axis=0) if vecs else np.zeros(Dk, dtype=np.float32)

def docvec_tfidf_kv7(tokens, tfidf_row):
    """
    TF-IDF-weighted pooling with pretrained vectors for ONE document:
      - Build {column → weight} from the doc's sparse TF-IDF row.
      - For tokens present in BOTH tfidf_lex7's vocab and kv's vocab,
        accumulate weight * vector, then divide by total weight Z.
      - Fall back to mean pooling if the row has no overlap.
    """
    # If this document's/query's TF-IDF row is empty (no in-vocab tokens),␣
↪return a zero vector.
    # This avoids divide-by-zero and gives a sane "no signal" embedding.
    if tfidf_row.nnz == 0:
        return np.zeros(Dk, dtype=np.float32)

    # Build a quick lookup: TF-IDF column index -> weight for THIS doc/query.
    # (tfidf_row.indices are the nonzero column ids; tfidf_row.data are the␣
↪corresponding weights.)
    col2w = {c: w for c, w in zip(tfidf_row.indices, tfidf_row.data)}

    # Accumulator for the weighted vector sum, and total weight Z.
    acc = np.zeros(Dk, dtype=np.float32); Z = 0.0

    # Iterate over tokens in this doc/query
    for t in tokens:
        # Find this token's column in the *frozen* TF-IDF vocabulary; None if␣
↪not a feature.
        col = term_to_col7.get(t)
        if col is not None:
            # Get the token's pretrained embedding (or None if OOV)
            v = kv_vec7(t)
            if v is not None:
                # Look up the TF-IDF weight for this token in THIS doc/query (0.
↪0 if absent)
                w = col2w.get(col, 0.0)
                if w > 0:
                    # Weighted sum of vectors and running total of weights
                    acc += w * v
                    Z   += w

    # Return the TF-IDF-weighted mean embedding if we accumulated any weight;
    # otherwise fall back to the plain mean-pooled embedding for stability.
    return acc / Z if Z > 0 else docvec_mean_kv7(tokens)


# Build both variants for the entire corpus.
```

```
X_gn_mean7   = np.vstack([docvec_mean_kv7(t) for t in TOKS_KV7])
X_gn_tfidf7  = np.vstack([docvec_tfidf_kv7(TOKS_KV7[i], Xtfidf_rows7[i]) for i␣
↪in range(len(TOKS_KV7))])

# L2-normalize rows so cosine similarity == dot product in retrieval.
X_gn_mean7_n  = _row_norm(X_gn_mean7)
X_gn_tfidf7_n = _row_norm(X_gn_tfidf7)
```

[ ]:
```
# 3) Unified comparison (uses ONLY the frozen tfidf_lex7 / X_tfidf_lex7)
#    One function to print top-k matches from TF-IDF (lexical), local CBOW, and␣
↪GoogleNews W2V.

def compare_search_v3(query, k=5,
                      show=("tfidf", "local_mean", "local_tfidf", "gn_mean",␣
↪"gn_tfidf")):
    q = str(query).strip()
    if not q:
        print("Empty query."); return
    print(f"\nQuery: {q}\n")

    # --- A) TF-IDF (lexical) ---
    # Always use the SAME fitted vectorizer as the index (tfidf_lex7) → dims␣
↪match by construction.
    if "tfidf" in show:
        q_tfidf = tfidf_lex7.transform([q.lower()])
        assert q_tfidf.shape[1] == Nlex7, f"Query TF-IDF dim {q_tfidf.shape[1]}␣
↪!= index dim {Nlex7}"
        sim_tf  = cosine_similarity(q_tfidf, X_tfidf_lex7)[0]  # cosine on␣
↪sparse is fine
        top_tf  = np.argsort(sim_tf)[-k:][::-1]
        print("TF-IDF (lexical) top-k:")
        for i in top_tf:
            print(" •", _fmt(i, float(sim_tf[i])))

    # --- B) Local CBOW ---
    # Tokenize query with the same tokenizer used to build the local matrices.
    if 'wv' in globals():
        q_tok_local = (tokenize(q) if 'tokenize' in globals() else analyzer7(q))

    # --- Local CBOW: mean-pooled query vs mean-pooled doc matrix ---
    if "local_mean" in show and X_loc_mean7_n is not None:
        # Build the query embedding by simple mean pooling over in-vocab tokens.
        vecs = [wv[t] for t in q_tok_local if t in wv.key_to_index]
        if vecs:
            qv = np.mean(vecs, axis=0)                  # mean of word vectors
```

```python
            qv /= (np.linalg.norm(qv) + 1e-12)          # L2-normalize → unit length
            sim = X_loc_mean7_n @ qv                    # cosine similarities (rows already L2-normalized)
            top = np.argsort(sim)[-k:][::-1]            # indices of top-k highest scores
            print("\nLocal CBOW (mean) top-k:")
            for i in top:
                print(" •", _fmt(i, float(sim[i])))

    # --- Local CBOW: TF-IDF-weighted query vs TF-IDF-weighted doc matrix ---
    if "local_tfidf" in show and X_loc_tfidf7_n is not None:
        # Use the SAME frozen TF-IDF (tfidf_lex7) to get the query's per-term weights.
        q_row = tfidf_lex7.transform([q.lower()])[0]    # sparse 1×V row for the query
        col2w = {c: w for c, w in zip(q_row.indices, q_row.data)}  # column -> TF-IDF weight

        # Accumulate a TF-IDF-weighted average of local CBOW vectors for the query.
        acc = np.zeros(Dloc, dtype=np.float32); Z = 0.0
        for t in q_tok_local:
            col = term_to_col7.get(t)                   # token's column in the frozen vocab (None if OOV for TF-IDF)
            if col is not None and t in wv.key_to_index:
                w = col2w.get(col, 0.0)                 # this query's TF-IDF weight for the token (0 if absent)
                if w > 0:
                    acc += w * wv[t]                    # weighted vector sum
                    Z   += w                            # total weight

        # If no positive weights overlapped, fall back to plain mean pooling (or zeros if no vectors).
        qv = (acc / Z) if Z > 0 else np.mean([wv[t] for t in q_tok_local if t in wv.key_to_index], axis=0)
        qv = np.zeros(Dloc, dtype=np.float32) if qv is None else qv

        # Normalize the query vector; then cosine == dot product with normalized doc matrix.
        qv /= (np.linalg.norm(qv) + 1e-12)
        sim = X_loc_tfidf7_n @ qv                       # cosine similarities to all docs
        top = np.argsort(sim)[-k:][::-1]                # top-k indices by score
        print("\nLocal CBOW (TF-IDF-weighted) top-k:")
        for i in top:
```

```
            print("  •", _fmt(i, float(sim[i])))


    # --- C) GoogleNews W2V ---
    # Tokenize query with the lexical analyzer for consistency with the pretrained path.
    q_tok_kv = analyzer7(q)

    if "gn_mean" in show:
        # Mean-pooled pretrained query embedding → cosine against mean-pooled doc matrix.
        vecs = [kv_vec7(t) for t in q_tok_kv]
        vecs = [v for v in vecs if v is not None]
        if vecs:
            qv = np.mean(vecs, axis=0)
            qv /= (np.linalg.norm(qv) + 1e-12)
            sim = X_gn_mean7_n @ qv
            top = np.argsort(sim)[-k:][::-1]
            print("\nW2V GoogleNews (mean) top-k:")
            for i in top:
                print("  •", _fmt(i, float(sim[i])))

    if "gn_tfidf" in show:
        # TF-IDF-weighted pretrained query embedding, using the SAME tfidf_lex7 vocabulary/weights.
        q_row = tfidf_lex7.transform([q.lower()])[0]
        col2w = {c: w for c, w in zip(q_row.indices, q_row.data)}
        acc = np.zeros(Dk, dtype=np.float32); Z = 0.0
        for t in q_tok_kv:
            col = term_to_col7.get(t)
            if col is not None:
                v = kv_vec7(t)
                if v is not None:
                    w = col2w.get(col, 0.0)
                    if w > 0:
                        acc += w * v; Z += w
        # Fallback to mean if the query has no TF-IDF overlap.
        qv = (acc / Z) if Z > 0 else np.mean([v for v in [kv_vec7(t) for t in q_tok_kv] if v is not None], axis=0)
        qv = np.zeros(Dk, dtype=np.float32) if qv is None else qv
        qv /= (np.linalg.norm(qv) + 1e-12)
        sim = X_gn_tfidf7_n @ qv
        top = np.argsort(sim)[-k:][::-1]
        print("\nW2V GoogleNews (TF-IDF-weighted) top-k:")
        for i in top:
            print("  •", _fmt(i, float(sim[i])))
```

```
# Quick sanity check: confirms index width and shows you the feature count.
print("Lexical TF-IDF index shape:", X_tfidf_lex7.shape)

# Examples:
# compare_search_v3("interest rates hike spooks investors", k=5)
# compare_search_v3("hurricane landfall florida panhandle", k=5)
# compare_search_v3("underdogs upset champions in penalty shootout", k=5)
# compare_search_v3("interest rates hike spooks investors", k=5,␣
 ↪show=("tfidf","gn_mean","local_mean"))
```

```
Lexical TF-IDF index shape: (1139, 53979)
```

Since the TF-IDF vectorizer was with with `ngram_range=(1,2)`, `min_df=2`, `max_df=0.7`, this means:

- Only 1-grams and 2-grams are included.
- They must appear in 2 docs (min_df=2) to avoid super-rare noise.
- They must appear in 70 % of docs (max_df=0.7) to drop stopword-level terms.

After applying those rules, the corpus (1139 docs) produced 53,979 distinct valid features.

It also means `X_tfidf_lex7[i, j]` is the TF-IDF weight of term `j` in document `i`.

[ ]:
```
queries = [
    "interest rates hike spooks investors",
    "hurricane landfall florida panhandle",
    "underdogs upset champions in penalty shootout",
]
for q in queries:
    compare_search_v3(q, k=5)
```

```
Query: interest rates hike spooks investors

TF-IDF (lexical) top-k:
 • 0.180  [Reuters] Dollar recovers some ground after payrolls blow, Yellen in
focus
 • 0.134  [Reuters] Don't know where U.S. stocks are headed? The options market
has a deal for you
 • 0.103  [Reuters] 'Abenomics' doubts drive foreigners off Japanese stocks,
volatility spikes
 • 0.100  [Reuters] World stocks tumble as Britain votes for EU exit
 • 0.087  [Reuters] Oil dips on dollar strength, Europe and Asia growth worries

Local CBOW (mean) top-k:
 • 0.702  [Reuters] Oil dips on dollar strength, Europe and Asia growth worries
 • 0.677  [Reuters] Twilio IPO exceeds expectations, despite Brexit angst
 • 0.673  [Reuters] U.S. banks flex capital muscle in annual stress test
 • 0.673  [Reuters] Brent crude tumbles to seven-week low on dollar rally,
```

Brexit turmoil
- 0.666 [Reuters] Rising rents, healthcare costs support U.S. underlying inflation

Local CBOW (TF-IDF-weighted) top-k:
- 0.846 [Reuters] Don't know where U.S. stocks are headed? The options market has a deal for you
- 0.837 [Reuters] Oil dips on dollar strength, Europe and Asia growth worries
- 0.834 [Reuters] U.S. banks flex capital muscle in annual stress test
- 0.832 [Reuters] Twilio IPO exceeds expectations, despite Brexit angst
- 0.829 [New York Times] The Fed Is Learning Just How Hard the Exit From Easy Money Will Be - The New York Times

W2V GoogleNews (mean) top-k:
- 0.609 [Reuters] 'Abenomics' doubts drive foreigners off Japanese stocks, volatility spikes
- 0.603 [New York Times] The Fed Is Learning Just How Hard the Exit From Easy Money Will Be - The New York Times
- 0.595 [New York Times] Fed Holds Interest Rates Steady and Plans Slower Increases - The New York Times
- 0.592 [Reuters] Don't know where U.S. stocks are headed? The options market has a deal for you
- 0.588 [Reuters] Dollar recovers some ground after payrolls blow, Yellen in focus

W2V GoogleNews (TF-IDF-weighted) top-k:
- 0.636 [New York Times] The Fed Is Learning Just How Hard the Exit From Easy Money Will Be - The New York Times
- 0.631 [New York Times] Fed Holds Interest Rates Steady and Plans Slower Increases - The New York Times
- 0.628 [Reuters] Rising rents, healthcare costs support U.S. underlying inflation
- 0.626 [Reuters] Dollar recovers some ground after payrolls blow, Yellen in focus
- 0.604 [New York Times] 'Brexit' Is Locking In the Forces That Already Haunt the Global Economy - The New York Times

Query: hurricane landfall florida panhandle

TF-IDF (lexical) top-k:
- 0.135 [Fox News] Tropical Storm Danielle swirls off Mexico's eastern coast
- 0.077 [New York Times] Anderson Cooper Covering Orlando Shooting With Touch of Empathy - The New York Times
- 0.063 [Fox News] East Coast on alert as Tropical Storm Colin forms, severe storm front moves through
- 0.057 [New York Times] $7 Million in Donations to Go Directly to Orlando Kin and Survivors - The New York Times
- 0.055 [Fox News] Gay-friendly beach towns, bars cautious in wake of Orlando

massacre

Local CBOW (mean) top-k:
• 0.644 [Fox News] Army Reserve officer killed in Orlando remembered as ’very positive young man’
• 0.638 [Fox News] New York airport security increased after Istanbul attack
• 0.632 [Fox News] Supreme Court leaves state assault weapons bans in place
• 0.624 [Fox News] LIVE BLOG: At least 50 killed in possible act of Islamic terror at Orlando nightclub
• 0.612 [Fox News] Eleven officers involved in gunfight that killed Orlando shooter hours after siege began

Local CBOW (TF-IDF-weighted) top-k:
• 0.856 [Fox News] Army Reserve officer killed in Orlando remembered as ’very positive young man’
• 0.825 [Reuters] Pride parades tinged with sadness after Orlando massacre
• 0.786 [Fox News] Rains slow, but flooding still threatens part of Texas
• 0.785 [Fox News] Man with weapons arrested in California ahead of Gay Pride parade, report says
• 0.783 [Fox News] Revelers: Gay pride events a victory over fear after Orlando

W2V GoogleNews (mean) top-k:
• 0.739 [Fox News] Tropical Storm Danielle swirls off Mexico’s eastern coast
• 0.631 [Fox News] East Coast on alert as Tropical Storm Colin forms, severe storm front moves through
• 0.543 [Fox News] Warnings and advisories issued across Central Texas amid new flooding concerns
• 0.515 [Fox News] Rains slow, but flooding still threatens part of Texas
• 0.480 [Reuters] Southern California wildfire spreads as blazes hit parched states

W2V GoogleNews (TF-IDF-weighted) top-k:
• 0.711 [Fox News] Tropical Storm Danielle swirls off Mexico’s eastern coast
• 0.627 [Fox News] East Coast on alert as Tropical Storm Colin forms, severe storm front moves through
• 0.533 [Fox News] Warnings and advisories issued across Central Texas amid new flooding concerns
• 0.469 [Fox News] Rains slow, but flooding still threatens part of Texas
• 0.425 [New York Times] West Virginia Floods Cause 23 Deaths and Vast Wreckage - The New York Times

Query: underdogs upset champions in penalty shootout

TF-IDF (lexical) top-k:
• 0.086 [New York Times] Lionel Messi and Argentina Miss Again as Chile Wins Copa América - The New York Times
• 0.084 [New York Times] U.S.G.A. Regrets ‘Distraction’ in Ruling Against

Dustin Johnson - The New York Times
- 0.081 [Fox News] Prosecutors seek death penalty for 2 suspects in doctor's murder, including her husband
- 0.061 [New York Times] U.S. Must Dig Deep to Stop Argentina's Lionel Messi - The New York Times
- 0.058 [New York Times] Two Defending Champions, but Two Outlooks, at Wimbledon - The New York Times

Local CBOW (mean) top-k:
- 0.696 [Fox News] Supreme Court leaves state assault weapons bans in place
- 0.690 [Fox News] Italian police capture fugitive mob boss sought for 20 years
- 0.688 [Reuters] Actor Anton Yelchin of 'Star Trek' films dies in freak accident
- 0.680 [Fox News] Veteran NPR journalist David Gilkey, translator killed in Afghanistan attack
- 0.680 [Fox News] Prosecutors seek death penalty for 2 suspects in doctor's murder, including her husband

Local CBOW (TF-IDF-weighted) top-k:
- 0.896 [Reuters] Led Zeppelin owes millions in royalties to musician: plaintiff attorney
- 0.881 [Fox News] Andrea Doria shipwreck more badly deteriorated than expected
- 0.880 [Fox News] Creator wins 148th Belmont Stakes in photo finish
- 0.878 [Fox News] After 7 decades, secret story of 'Nazi Titanic' is told
- 0.874 [Fox News] 500 year-old shipwreck loaded with gold found in Namibian desert

W2V GoogleNews (mean) top-k:
- 0.602 [New York Times] Lionel Messi and Argentina Miss Again as Chile Wins Copa América - The New York Times
- 0.592 [New York Times] Penguins Finish Off Sharks to Win Stanley Cup - The New York Times
- 0.582 [New York Times] Warriors Edge Thunder to Extend Dream Season to N.B.A. Finals - The New York Times
- 0.582 [New York Times] Cavaliers Defeat Warriors to Win Their First N.B.A. Title - The New York Times
- 0.575 [New York Times] N.B.A. Finals: How the Warriors and Cavaliers Match Up - The New York Times

W2V GoogleNews (TF-IDF-weighted) top-k:
- 0.559 [New York Times] U.S.G.A. Regrets 'Distraction' in Ruling Against Dustin Johnson - The New York Times
- 0.535 [New York Times] Golden State Warriors Slipped, Then Fell, Despite a Record Season - The New York Times
- 0.534 [Reuters] Muguruza dethrones Serena again in Paris to win French Open
- 0.522 [New York Times] Game 7 of N.B.A. Finals Draws Close to 31 Million

```
Viewers - The New York Times
 • 0.520  [New York Times] Cavaliers Defeat Warriors to Win Their First N.B.A.
Title - The New York Times
```

4.4 8) Semantic Search — TF-IDF (lexical) vs Sentence-BERT (SBERT) (semantic)

This is a demo of using transformers, which creates contextual (non static) embeddings. We haven't covered them yet so this is just for interest.

Builds two search indexes over the corpus:

1. a TF-IDF index that matches on shared words/phrases (lexical overlap), and
2. an SBERT embedding index that matches on meaning (even with different wording).

We call compare_search("your query", k=5) to see the top-k results from both systems side-by-side.

This is a higher-level demo showing how transformer embeddings capture deep semantics beyond word vectors.

- TF-IDF excels when the query shares surface words with the documents.
- SBERT shines when it doesn't — handling paraphrases, synonyms, and context far better than n-gram or Word2Vec models.

[ ]:
```python
# === 8) Semantic Search - TF-IDF (lexical) vs SBERT (semantic) ===
# Call: compare_search("interest rates hike spooks investors", k=5)

# (1) Build TF-IDF index (lexical) - reuses STOPWORDS/TOK from earlier
tfidf = TfidfVectorizer(
    ngram_range=(1,2),
    min_df=2,
    max_df=0.7,
    stop_words=STOPWORDS,
    token_pattern=TOK,
    sublinear_tf=True
)
X_tfidf = tfidf.fit_transform(DOCS)

# (2) Build SBERT index (semantic) with chunking (avoid truncation)
if 'enc' not in globals() or enc is None:
    try:
        from sentence_transformers import SentenceTransformer
        import torch
        device = "cuda" if torch.cuda.is_available() else "cpu"
        enc = SentenceTransformer("all-MiniLM-L6-v2", device=device)
    except Exception as e:
        enc = None
        print("SBERT unavailable →", e)

# Define chunker once (idempotent)
```

```python
if 'chunk_text' not in globals():
    def chunk_text(t, max_words=250):
        ws = str(t).split()
        return [" ".join(ws[i:i+max_words]) for i in range(0, len(ws),␣
↪max_words)] or [""]

# Fresh helper to avoid name clashes with any earlier encode_docs
def sbert_encode_docs(texts, encoder, bs=128):
    """Encode each doc by averaging normalized embeddings of its chunks."""
    if encoder is None:
        return None
    embs = []
    for doc in texts:
        chunks = chunk_text(doc)
        E = encoder.encode(
            chunks,
            batch_size=bs,
            normalize_embeddings=True,
            convert_to_numpy=True,
            show_progress_bar=False
        )
        embs.append(E.mean(axis=0))
    return np.vstack(embs)

X_sbert = sbert_encode_docs(DOCS, enc, bs=128)


# (3) Compare search: TF-IDF vs SBERT
def compare_search(query, k=5):
    """Print top-k nearest docs for a free-text query using TF-IDF and SBERT."""
    q = str(query).strip()
    if not q:
        print("Empty query."); return
    print(f"\nQuery: {q}\n")

    # TF-IDF cosine (lexical overlap)
    q_tfidf = tfidf.transform([q.lower()])
    sim_tf = cosine_similarity(q_tfidf, X_tfidf)[0]
    top_tf = np.argsort(sim_tf)[-k:][::-1]
    print("TF-IDF (lexical) top-k:")
    for i in top_tf:
        print(" •", _fmt(i, float(sim_tf[i])))

    # SBERT cosine (dot product; embeddings normalized)
    if X_sbert is not None and enc is not None:
        q_emb = enc.encode([q], normalize_embeddings=True,␣
↪convert_to_numpy=True)[0]
```

```
        sim_se = X_sbert @ q_emb
        top_se = np.argsort(sim_se)[-k:][::-1]
        print("\nSBERT (semantic) top-k:")
        for i in top_se:
            print(" •", _fmt(i, float(sim_se[i])))
    else:
        print("\nSBERT (semantic) top-k: [index unavailable]")
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94:
UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab
(https://huggingface.co/settings/tokens), set it as secret in your Google Colab
and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access
public models or datasets.
  warnings.warn(

modules.json:   0%|          | 0.00/349 [00:00<?, ?B/s]

config_sentence_transformers.json:   0%|          | 0.00/116 [00:00<?, ?B/s]

README.md: 0.00B [00:00, ?B/s]

sentence_bert_config.json:   0%|          | 0.00/53.0 [00:00<?, ?B/s]

config.json:   0%|          | 0.00/612 [00:00<?, ?B/s]

model.safetensors:   0%|          | 0.00/90.9M [00:00<?, ?B/s]

tokenizer_config.json:   0%|          | 0.00/350 [00:00<?, ?B/s]

vocab.txt: 0.00B [00:00, ?B/s]

tokenizer.json: 0.00B [00:00, ?B/s]

special_tokens_map.json:   0%|          | 0.00/112 [00:00<?, ?B/s]

config.json:   0%|          | 0.00/190 [00:00<?, ?B/s]
```

[ ]:

```
queries = [
    "interest rates hike spooks investors",
    "hurricane landfall florida panhandle",
    "underdogs upset champions in penalty shootout",
]
for q in queries:
    compare_search(q, k=5)
```

```
Query: interest rates hike spooks investors

TF-IDF (lexical) top-k:
```

- 0.180 [Reuters] Dollar recovers some ground after payrolls blow, Yellen in focus
- 0.134 [Reuters] Don't know where U.S. stocks are headed? The options market has a deal for you
- 0.103 [Reuters] 'Abenomics' doubts drive foreigners off Japanese stocks, volatility spikes
- 0.100 [Reuters] World stocks tumble as Britain votes for EU exit
- 0.087 [Reuters] Oil dips on dollar strength, Europe and Asia growth worries

SBERT (semantic) top-k:
- 0.484 [New York Times] The Fed Is Learning Just How Hard the Exit From Easy Money Will Be - The New York Times
- 0.414 [Reuters] 'Abenomics' doubts drive foreigners off Japanese stocks, volatility spikes
- 0.397 [New York Times] Fed Holds Interest Rates Steady and Plans Slower Increases - The New York Times
- 0.380 [New York Times] Central Banks Worry About Engaging World Markets After 'Brexit' - The New York Times
- 0.379 [Reuters] Exclusive: Ousted CEO Laplanche studies LendingClub takeover - sources

Query: hurricane landfall florida panhandle

TF-IDF (lexical) top-k:
- 0.135 [Fox News] Tropical Storm Danielle swirls off Mexico's eastern coast
- 0.077 [New York Times] Anderson Cooper Covering Orlando Shooting With Touch of Empathy - The New York Times
- 0.063 [Fox News] East Coast on alert as Tropical Storm Colin forms, severe storm front moves through
- 0.057 [New York Times] $7 Million in Donations to Go Directly to Orlando Kin and Survivors - The New York Times
- 0.055 [Fox News] Gay-friendly beach towns, bars cautious in wake of Orlando massacre

SBERT (semantic) top-k:
- 0.547 [Fox News] Tropical Storm Danielle swirls off Mexico's eastern coast
- 0.351 [Fox News] Disney rep says company plans to 'thoroughly review' alligator signage after attack
- 0.337 [Fox News] Body of 2-year-old boy snatched by alligator recovered, sheriff confirms
- 0.325 [New York Times] Divers Find Body of Toddler Snatched by Alligator at Disney Resort - The New York Times
- 0.317 [Reuters] Disney to post alligator warning signs after boy's death

Query: underdogs upset champions in penalty shootout

TF-IDF (lexical) top-k:
- 0.086 [New York Times] Lionel Messi and Argentina Miss Again as Chile Wins

Copa América - The New York Times

- 0.084 [New York Times] U.S.G.A. Regrets ‘Distraction’ in Ruling Against Dustin Johnson - The New York Times
- 0.081 [Fox News] Prosecutors seek death penalty for 2 suspects in doctor’s murder, including her husband
- 0.061 [New York Times] U.S. Must Dig Deep to Stop Argentina’s Lionel Messi - The New York Times
- 0.058 [New York Times] Two Defending Champions, but Two Outlooks, at Wimbledon - The New York Times

SBERT (semantic) top-k:

- 0.365 [New York Times] Lionel Messi and Argentina Miss Again as Chile Wins Copa América - The New York Times
- 0.321 [Reuters] Euro 2016 violence spreads to second French city
- 0.317 [New York Times] Warriors Edge Thunder to Extend Dream Season to N.B.A. Finals - The New York Times
- 0.312 [New York Times] Warriors, Resilient at Home, Cruise Against the Cavaliers - The New York Times
- 0.306 [New York Times] U.S. Must Dig Deep to Stop Argentina’s Lionel Messi - The New York Times