Bag-of-Words Text Classification with Logistic Regression

Greg Baker

Week 10 — 14th October 2025

Where we are heading today

- Quick reset: why classic NLP still matters in 2025.
- Build a spam filter with scikit-learn's `LogisticRegression`.
- Compare count vectors and TF–IDF weighting.
- Talk about evaluation, pitfalls, and when this toolkit shines (or doesn't).
- Flag how this slots into Assessment 2 ideas.

Why bag-of-words?

- Dead simple representation: one dimension per token, ignore word order.
- Fast to train, easy to interpret, tiny carbon footprint compared with LLMs.
- Strong baseline for document classification, search, quick prototypes.
- Still underpins many production spam filters and support-ticket routers.
- Downsides: no context, brittle to typos, vocabulary can explode without care.

Dataset for our demo

- We'll use the "SMS Spam" dataset hosted on OpenML (5,574 text messages).
- Labels: `ham` (legit) vs `spam`.
- Light cleaning: trim whitespace, normalise casing, leave punctuation so the vectoriser can decide.
- Split into train/validation/test (e.g. 60%/20%/20%).
- If you want emails instead, swap in the Enron corpus — same pipeline, just slower.

Loading and splitting the data

```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

data = fetch_openml("sms_spam", version=1,
    as_frame=True)
X = data.data["text"].str.strip()
y = data.target

X_train, X_temp, y_train, y_temp = train_test_split(
    X, y, test_size=0.4, stratify=y, random_state=42
)
X_valid, X_test, y_valid, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, stratify=y_temp,
        random_state=42
)
```

CountVectorizer refresher

- Tokenises text (default: whitespace + punctuation heuristics).
- Builds vocabulary of most common tokens `max_features`.
- Produces sparse matrix of raw counts.
- Options we will poke: `ngram_range`, `stop_words`, `min_df`.
- Great for quick baselines and interpretability (top words per class).

Count vector pipeline with logistic regression

```
from sklearn.feature_extraction.text import
    CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

count_logreg = make_pipeline(
    CountVectorizer(max_features=2000,
        ngram_range=(1, 2)),
    LogisticRegression(max_iter=1000)
)
count_logreg.fit(X_train, y_train)
print(f"Validation accuracy:
    {count_logreg.score(X_valid, y_valid):.3f}")
```

Interpreting the coefficients

- Logistic regression assigns a weight per token: positive pushes towards spam.
- Extract the vectoriser vocabulary, align with `coef_` values.
- Handy for quick sanity checks ("win", "unsubscribe" should look spammy).
- Also reveals leakage — if you see phone numbers or "ham" tokens ranked highly, revisit preprocessing.

When raw counts fall short

- Frequent tokens dominate counts even if they are boring stop-words.
- Longer documents produce larger totals, biasing the classifier.
- TF–IDF rescales counts by how distinctive a term is across the corpus.
- Works nicely with linear models by keeping features in comparable ranges.

TF–IDF pipeline

```
from sklearn.feature_extraction.text import
    TfidfVectorizer

tfidf_logreg = make_pipeline(
    TfidfVectorizer(max_features=4000,
        ngram_range=(1, 2),
                    sublinear_tf=True, norm="l2"),
    LogisticRegression(max_iter=1000, C=2.0)
)

tfidf_logreg.fit(X_train, y_train)
print(f"Validation accuracy:
    {tfidf_logreg.score(X_valid, y_valid):.3f}")
```

Comparing vectorisers

Count vectors

- Raw token counts.
- Sensitive to document length.
- Easier to reason about feature impact.
- Strong when vocabulary is well curated.

TF–IDF

- Down-weights ubiquitous tokens automatically.
- Normalises document length.
- Often lifts performance a couple of points.
- Slightly harder to explain to stakeholders, but still interpretable.

Confusion matrix and reports

```
from sklearn.metrics import ConfusionMatrixDisplay,
    classification_report

ConfusionMatrixDisplay.from_estimator(tfidf_logreg,
    X_test, y_test,
                                      display_labels=["h
                                          "spam"],
                                      cmap="Blues")
print(classification_report(y_test,
    tfidf_logreg.predict(X_test)))
```

- Track precision/recall: false positives annoy users, false negatives let spam through.
- Calibrate threshold (`predict_proba`) if business costs are asymmetric.

Peeking inside predictions

- Use `tfidf_logreg.named_steps['logisticregression'].coef_` for feature weights.
- `peek_inside.py` extracts the top ham/spam tokens and plots their weights.
- Show the chart alongside misclassified examples to expose blind spots.
- Nice segue into fairness discussions (e.g. different language styles).

Visualising influential tokens

```
    top_spam_idx =
        np.argsort(weights)[-top_n:][::-1]
    top_ham_idx = np.argsort(weights)[:top_n]

    tokens =
        np.concatenate([feature_names[top_ham_idx],
        feature_names[top_spam_idx]])
    strengths =
        np.concatenate([weights[top_ham_idx],
        weights[top_spam_idx]])
    classes: Iterable[str] = ["ham"] * top_n +
        ["spam"] * top_n

    return pd.DataFrame({"token": tokens,
        "weight": strengths, "class": classes})
```

Green flags

- You have thousands to hundreds of thousands of short or medium texts.
- Labels are tidy and well balanced (or you can weight classes).
- Vocabulary is mostly "literal" — marketing emails, support tickets, news headlines.
- Need fast iteration, easy deployment, or explainability.
- Limited compute budget (fits on a laptop, retrains in seconds).

Red flags

- Tasks needing deep semantic understanding (sarcasm, legal reasoning, code synthesis).
- Long documents where word order and discourse matter.
- Domains with heavy obfuscation or adversaries that constantly mutate tokens.
- Multilingual data with non-Latin scripts unless you customise tokenisation.
- Tiny datasets (risk of overfitting) or huge label sets (feature sparsity hurts).

Where to go next

- Try character n-grams for noisy text (catch “fr33” and “vi@gra”).
- Add simple engineered features: message length, number of URLs, sender domain.
- Compare with Naive Bayes, linear SVM, or a small neural net for context.
- Cross-validate `C`, `max_features`, and `ngram_range` once the pipeline works.
- Eventually graduate to contextual embeddings (e.g. `sentence-transformers`) when baseline taps out.

Key takeaways

- Bag-of-words remains a powerful starting point for NLP classification.
- Logistic regression + TF–IDF gives robust, interpretable spam detection.
- Evaluation matters: focus on costs of mistakes, not just accuracy.
- Know when to keep it simple and when to escalate to richer models.
- Feed these ideas into Assessment 2: build a pipeline, justify your feature choices, compare against a dummy.

Next steps

- Practical this week: hands-on with scikit-learn text pipelines.
- Bring any Assessment 2 dataset ideas to the workshop — we can sanity check feature plans.
- Reading: Chapter 6 of the course notes (bag-of-words) + scikit-learn docs on `TfidfVectorizer`.
- Questions? Drop them on Ed, reach out to tutors, or grab me after class.