COMP2200/COMP6200 Practical Exercise (Week 10): Vectorisers, Limits, and Logistic Stories

School of Computing

Preparation

Choose your environment: either run locally with `uv` or use Google Colab.

1. **Local with `uv`**:
   (a) Install `uv`.
   MacOS/Linux: `curl -LsSf https://astral.sh/uv/install.sh | sh`
   Windows: `powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"`
   (b) Create a folder for this practical (for example, `week10-prac`) and open a terminal in it.
   (c) Set up your environment:

   ```
   uv init
   uv add pandas scikit-learn matplotlib seaborn umap-learn
   ```

   (d) Launch Jupyter with `uv run --with jupyter jupyter lab` (or `... notebook`).
2. **Google Colab**:
   (a) Open Colab and create a new notebook.
   (b) Upload `enron_practical_sample.csv` from iLearn or this week's zip file.
   (c) Install packages with `!pip install pandas scikit-learn matplotlib seaborn umap-learn`.

Load the CSV with:

```python
import pandas as pd
emails = pd.read_csv("enron_practical_sample.csv")
```

Because we are going to run many different models and choose between them, we do need to keep some hold-out data for reporting our final accuracy. Create a train/test split using `train_test_split`. Use the train data for everything until the end of Part C.

Double-check that you did include **umap-learn** in the instructions above. You'll need it in Part B.

Part A — Tokenisation tuning lab (20 min)

Starting with this:

```python
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
vectoriser = TfidfVectorizer(stop_words="english", ngram_range=(1, 1))
X_train = vectoriser.fit_transform(train_text)
```

1. Record the vocabulary size and the top 10 weighted terms (`vectoriser.idf_` or `vectoriser.vocabulary_`). Here's how to extract and display them:

```python
# Vocabulary size
vocab_size = len(vectoriser.vocabulary_)
print(f"Vocabulary size: {vocab_size}")

# Top 10 weighted terms (highest IDF scores)
idf_series = pd.Series(vectoriser.idf_,
                       index=vectoriser.get_feature_names_out())
top_10_terms = idf_series.nlargest(10)
print("\nTop 10 weighted terms:")
print(top_10_terms)
```

2. Rotate through the following tweaks, keeping notes on what changes and why:
   - **Character n-grams:** switch to `analyzer="char_wb"` with `ngram_range=(3,5)`.
   - **Frequency trims:** set `min_df=2` and `max_df=0.8`.
   - **Bigrams:** use `ngram_range=(1,2)` with default word analyzer.
3. Highlight one surprise: what token appeared/disappeared, and how could that affect logistic regression weights?

The aim is to give every group a talking point for the main modelling challenge.

Part B — UMAP visualisation (20 min)

Before diving into classification, let's visualise the email landscape to see if spam and ham naturally cluster.

1. Create a TF-IDF representation of all emails (train + test combined) with a reasonable feature limit:

```python
from sklearn.feature_extraction.text import TfidfVectorizer
import umap
import matplotlib.pyplot as plt
import seaborn as sns

# Combine all text for visualization
all_text = pd.concat([train_text, test_text])
all_labels = pd.concat([train_labels, test_labels])

# Vectorize with moderate feature budget
vectoriser = TfidfVectorizer(max_features=1000, stop_words="english")
X_tfidf = vectoriser.fit_transform(all_text)

# Reduce to 2D with UMAP
reducer = umap.UMAP(n_components=2, random_state=2025, n_neighbors=15)
embedding = reducer.fit_transform(X_tfidf)
```

2. Create a scatter plot coloring points by spam/ham. Use `alpha=0.5` for transparency:

```python
plt.figure(figsize=(10, 8))
sns.scatterplot(x=embedding[:, 0], y=embedding[:, 1],
                hue=all_labels, palette=["#2ecc71", "#e74c3c"],
                alpha=0.5, s=30)
plt.title("UMAP Projection of Enron Emails (TF-IDF)")
plt.xlabel("UMAP 1")
plt.ylabel("UMAP 2")
plt.legend(title="Label")
plt.show()
```

3. Are spam and ham visually separable? Where do the classes overlap? What does this suggest about the difficulty of the classification task?

4. **Optional:** Try different `n_neighbors` values (5, 15, 50) and compare how the structure changes. UMAP's neighborhood parameter controls local vs. global structure emphasis.

5. **Optional:** Try different vectorisation options (from Part A) and see if that makes a difference

Part C — Feature budget challenge (40 min)

In this round you design the leanest email classifier that still performs. You may only keep **1,000 features**.

1. Start from this skeleton pipeline and plug in your favourite vectoriser from Part A:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_validate, StratifiedKFold
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ("vectoriser", TfidfVectorizer(max_features=1000, stop_words="english")),
    ("model", LogisticRegression(max_iter=500))
])

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=2025)
metrics = cross_validate(pipeline, train_text, train_labels,
                         cv=cv, scoring=["accuracy", "f1_macro"],
                         return_train_score=True)
```

2. Iterate on preprocessing knobs (n-gram ranges, document frequency limits, sublinear tf) while respecting the 1,000-feature budget. Use `metrics["test_f1_macro"].mean()` as the scoreboard.

3. **Optional:** Instead of 1000, how small can you make it your model? `TfidfVectorizer` can take a `vocabulary=` argument where you can specify exactly which words or n-grams that it is allowed to use.

4. Once satisfied, fit the pipeline on the training data and evaluate on the held-out test data. Report accuracy, macro-F1, and the confusion matrix.

5. Document your best configuration in the shared spreadsheet: vectoriser settings, mean cross-validated F1, and one sentence explaining why it works.

Part D — Regularisation and coefficient storytelling (30 min)

Next we unpack what the logistic regression weights mean and how regularisation changes them.
Smaller C = stronger regularisation (smaller coefficients).

1. Record the vectoriser settings that delivered your best Part C score, then extend your pipeline to expose coefficients:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.utils import Bunch

best_params = {
    "max_features": 1000,
    "stop_words": "english",
    # add any other parameters that worked best in Part C
}
```

```
def train_with_c(C=1.0, penalty="l2") -> Bunch:
    pipeline = Pipeline([
        ("vectoriser", TfidfVectorizer(**best_params)),
        ("model", LogisticRegression(max_iter=1000, C=C, penalty=penalty,
                                     solver="liblinear" if penalty == "l1" else "lbfgs"))
    ])
    pipeline.fit(train_text, train_labels)
    return Bunch(
        pipeline=pipeline,
        coefficients=pipeline.named_steps["model"].coef_[0],
        vocab=pipeline.named_steps["vectoriser"].get_feature_names_out()
    )
```

2. Run the helper for $C \in \{0.1, 1, 10\}$ with L2 and for $C = 1$ with L1. Plot coefficient magnitudes (you could do this in Excel, or you could make a pandas Series and plot it)
3. For your winning configuration, list the top 5 positive and negative tokens. Translate them into plain language rules ("Emails mentioning *meeting* push the score toward a particular class").

Part E (optional) — Error clinic and moderation huddle (10 min)

To close, we stress-test the model against borderline cases.

1. Generate a DataFrame of misclassifications on the test split with columns `text`, `true_label`, `predicted`, and `probability` (the predicted class probability).
2. Tag each misclassified email with the likely cause: **ambiguous tone**, **missing vocabulary**, or **label noise**. Capture ideas for new features or thresholds in a shared doc.

Part F – Quiz questions (10 min)

Spend some time on the Quiz questions in anticipation of Assignment 2d.