REP or DEM LSTM classification report

Paul Elvis Otto
Duke University / Hertie School

Table of contents

1 Intro

Classifying short social-media texts is a standard NLP task and a useful testbed for comparing lexical and sequence-based models. This project predicts whether a tweet posted by a member of the 112th U.S. Congress was authored by a Democrat or a Republican. The dataset, sourced from Harvard Dataverse, contains each post's text as well as metadata on the author's party and chamber (House vs. Senate).

The main model is a bidirectional LSTM classifier operating over contextual token embeddings. Instead of learning word representations from scratch, the pipeline uses a pretrained MiniLM Transformer as a frozen feature extractor (loaded via the MLX-Embeddings/Hugging Face adaptation). The BiLSTM and linear classification head are trained on top of these fixed embeddings. As a transparent baseline, a Multinomial Naive Bayes model with TF–IDF n-gram features is implemented on the same train/validation/test split, and its performance is compared to the BiLSTM in the empirical assessment section.

To situate the task, an exploratory data analysis is presented in Section 3. The text preprocessing and label normalization used to construct the modeling dataset are described in Section 2.

2 Data Preparation

To ensure the data is usable and not influenced by elements such as dates, mentions, or links, each post was cleaned using a set of regular expressions. The cleaning script is available in the model's repository. The cleaned dataset excludes the following:

- Dates
- Mentions
- Hashtags
- URLs

The data is read in already cleaned into the model pipeline.

3 EDA

This section provides an initial overview of the dataset by examining its basic structure and several descriptive metrics.

The dataset has a dimension of 334606, 5. To begin, we look at how posts are distributed across chambers and parties. The overall distribution is visualized in Figure 1.

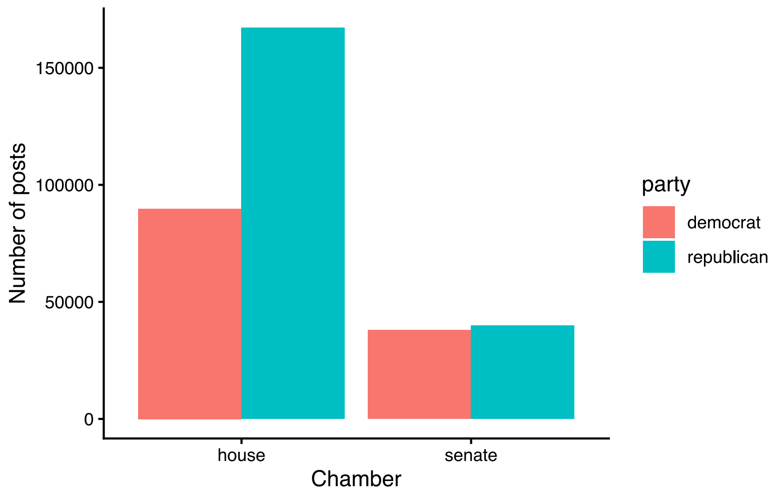


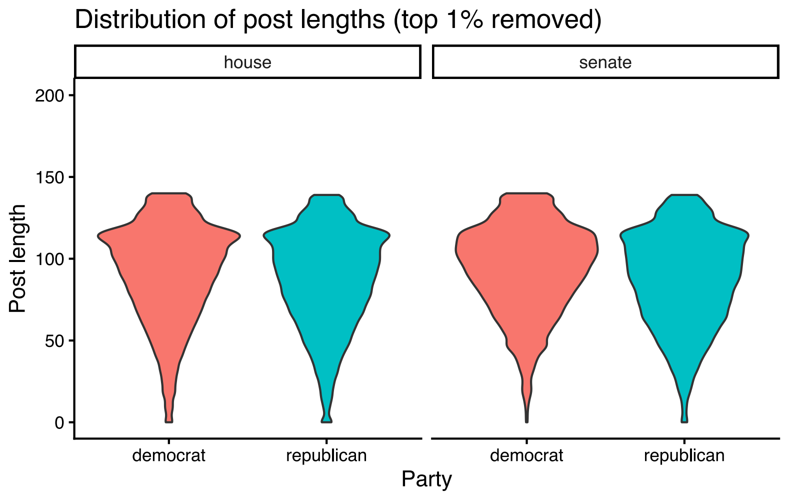
Figure 1: Distribution of posts by chamber and party

A more detailed breakdown of total posts per chamber and party is provided in the table below:

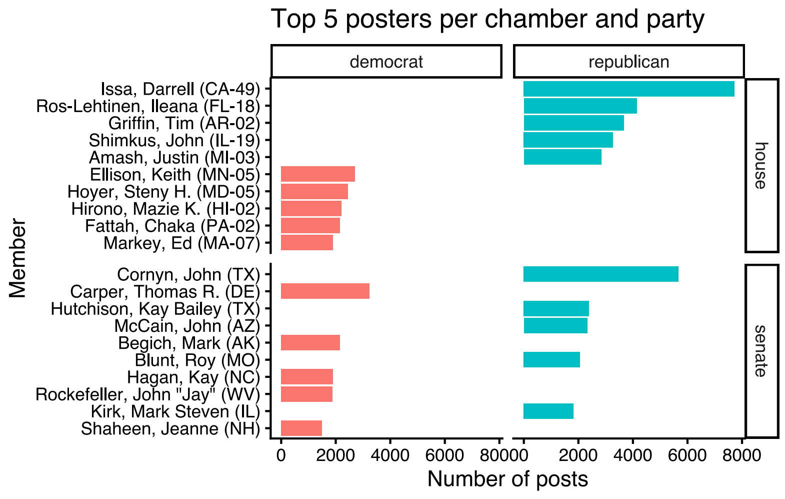
Number of Posts by Chamber and Party

	democrat	republican
house	89808	167004
senate	37890	39904

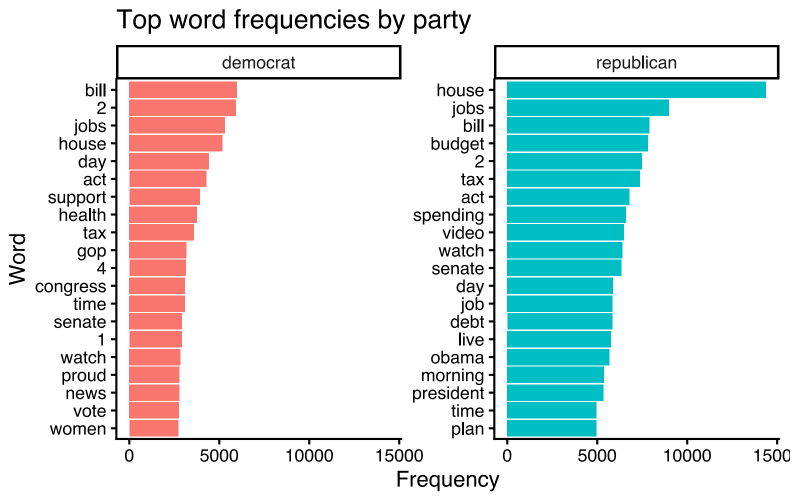
Next, the distribution of post lengths is examined. To limit the influence of outliers, the top one percent of longest posts are excluded:



The dataset also allows identifying the most active members. The figure below displays the top five posters for each chamber–party combination:



Finally, the most frequent words used by each party are explored. After tokenizing posts, removing stopwords, and excluding a small set of custom stop terms, the top twenty terms per party are identified:



4 Model specification

4.1 Tokenization and feature extraction

The input pipeline leverages a pre-trained Transformer model, specifically `all-MiniLM-L6-v2`, to generate contextualized semantic features. Unlike traditional pipelines that feed token IDs directly to the classifier, this architecture utilizes the Transformer as a frozen feature extractor.

Raw text is tokenized and padded to a fixed sequence length $T = 128$. Let $\mathbf{t}$ be the sequence of token identifiers. These are passed through the quantized (4-bit) Transformer model $\mathcal{F}$ to yield a sequence of dense vectors:

$$\mathbf{X} = \mathcal{F}(\mathbf{t}) \in \mathbb{R}^{T\times D}, \tag{1}$$

where D is the embedding dimension (inferred from the Transformer output, typically 384 for MiniLM). Crucially, a stop-gradient operation is applied to $\mathbf{X}$:

$$\mathbf{X}_{\text{fixed}} = \text{StopGradient } (\mathbf{X}), \tag{2}$$

ensuring that gradients are not backpropagated into the Transformer layers, treating the embeddings as static inputs to the downstream LSTM.

4.2 Bidirectional LSTM encoder

A custom implementation of a Bidirectional LSTM is employed to model the temporal dependencies within the sequence of embeddings $\mathbf{X}_{\text{fixed}}$.

The architecture consists of two independent LSTM layers: a forward pass ($\overrightarrow{\text{LSTM}}$) and a backward pass ($\overleftarrow{\text{LSTM}}$). For a single direction, the state update at timestep t, given input $\mathbf{x}_t$ and previous hidden state $\mathbf{h}_{t-1}$, is governed by the standard gate equations:

$$\begin{aligned}
\mathbf{i}_t &= \sigma(W_i\mathbf{x}_t + U_i\mathbf{h}_{t-1} + \mathbf{b}_i) \\
\mathbf{f}_t &= \sigma\left(W_f\mathbf{x}_t + U_f\mathbf{h}_{t-1} + \mathbf{b}_f\right) \\
\mathbf{o}_t &= \sigma(W_o\mathbf{x}_t + U_o\mathbf{h}_{t-1} + \mathbf{b}_o) \\
\mathbf{g}_t &= \tanh\left(W_g\mathbf{x}_t + U_g\mathbf{h}_{t-1} + \mathbf{b}_g\right) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
\end{aligned} \tag{3}$$

where σ is the sigmoid function and $\odot$ is the Hadamard product. The hidden state dimension is $H = 128$. The backward LSTM processes the sequence in reverse order. The final representation at each timestep is the concatenation of both directional states:

$$\mathbf{h}_t = \left[\overrightarrow{\mathbf{h}}_t; \overleftarrow{\mathbf{h}}_t\right] \in \mathbb{R}^{2H}. \tag{4}$$

4.3 Masked mean pooling

To handle the padding artifacts present in the fixed-length sequences, a masked mean pooling operation is applied. Let $\mathbf{m} \in \{0, 1\}^T$ be the attention mask where 1 denotes a valid token. The fixed-size sentence representation $\mathbf{z}$ is computed as:

$$\mathbf{z} = \frac{\sum_{t=1}^{T} m_t\mathbf{h}_t}{\sum_{t=1}^{T} m_t + \epsilon} \in \mathbb{R}^{2H}, \tag{5}$$

where $\epsilon = 10^{-6}$ ensures numerical stability. This collapses the temporal dimension, resulting in a single vector capturing the global context of the post.

4.4 Classification head

The pooled vector $\mathbf{z}$ serves as the input to the classification head. Regularization is applied via Dropout with probability $p = 0.1$:

$$\tilde{\mathbf{z}} = \text{Dropout } (\mathbf{z}). \tag{6}$$

A linear projection layer maps the features to the unnormalized logits for the two classes:

$$\mathbf{o} = W_{out}\tilde{\mathbf{z}} + \mathbf{b}_{out} \in \mathbb{R}^2. \tag{7}$$

The predicted class $\hat{y}$ is derived via the argmax of the logits.

4.5 Training objective

The model is trained using the Cross-Entropy loss function. For a batch of N samples, the objective is to minimize:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} \text{CrossEntropy } \left(\mathbf{o}^{(i)}, y^{(i)}\right). \tag{8}$$

Optimization is performed using the **Adam** optimizer with a learning rate $\eta = 10^{-3}$. The training step, including the loss calculation and gradient update, is Just-In-Time (JIT) compiled into a comprehensive computation graph using `mx.compile` to maximize execution speed.

4.6 Computational considerations

The implementation is optimized for Apple Silicon using the MLX framework. Several specific strategies are employed for efficiency:

1. **Lazy Evaluation & Materialization:** MLX uses lazy evaluation. To prevent the computation graph from growing indefinitely during the iterative data loading process, `mx.eval` is explicitly called on batch inputs and loss values to force materialization.
2. **Quantized Feature Extraction:** The embedding model (`all-MiniLM-L6-v2`) utilizes 4-bit quantization, significantly reducing memory bandwidth requirements during the feature extraction phase.
3. **Precision:** The embeddings are cast to `float16` (`mx.float16`) before entering the LSTM, halving the memory footprint of the batch tensor compared to `float32` and leveraging the hardware's native half-precision performance.

5 Data Pipeline

The pipeline proceeds as:

- Read the cleaned CSV (`congress_complete.csv`) and keep the relevant columns (`post` and `party`).
- Drop rows with missing text or party labels.
- Normalize party strings and map them to binary labels (Democrat = 0, Republican = 1), discarding any other labels.

- Perform a stratified train/validation/test split with proportions 80/10/10 using a fixed random seed.
- During training and evaluation, tokenize **on the fly** per batch and pad/truncate to a fixed length (T=128).
- For each minibatch, pass token IDs and attention masks through a **frozen, quantized MiniLM encoder** to obtain contextual token embeddings. Gradients are stopped at this stage, so only the downstream BiLSTM is trained.
- Cast embeddings to `float16` to reduce memory use; keep attention masks to zero out padding tokens before recurrence and for masked mean pooling.

This design avoids storing large precomputed embedding tensors in RAM. Instead, embeddings are computed batch-wise and materialized with `mx.eval` so MLX graphs do not accumulate across iterations.

5.1 Training Loop

The training routine runs for 10 epochs with a fixed batch size of 64. In each epoch:

- Set the model to training mode.
- Iterate over batches produced by `batch_iterate_text(..., drop_last=True)` so every training step has identical shape (required for `mx.compile`).
- For each batch:
  - Compute logits with the BiLSTM classifier.
  - Compute mean cross-entropy loss.
  - Backpropagate to obtain gradients for the BiLSTM parameters.
  - Update parameters using the **Adam** optimizer with learning rate (10^{-3}).
- After each epoch, switch to evaluation mode and compute validation loss and accuracy over the full validation split (`drop_last=False`).

The training step is JIT-compiled via `mx.compile` for speed. There is no checkpointing or best-model reload in the current code; the reported test results come from the final epoch's parameters.

5.2 Implementation in Code

The biLStM model has been in apples mlx framework to enable a bare metal run of the model on apple silicon hardware. The implementation has ben performd on top of the mlx framework provided `nn.Module` function.

for that a LSTM cell has been set up as follows

~~~
```{python}
class LSTMCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.xh_to_gates = nn.Linear(input_size + hidden_size, 4 *
~~~

~~~
hidden_size)

    def __call__(self, x_t, state):
        h_prev, c_prev = state
        xh = mx.concatenate([x_t, h_prev], axis=-1)
        gates = self.xh_to_gates(xh)
        i, f, o, g = mx.split(gates, 4, axis=-1)

        i = mx.sigmoid(i)
        f = mx.sigmoid(f)
        o = mx.sigmoid(o)
        g = mx.tanh(g)

        c_t = f * c_prev + i * g
        h_t = o * mx.tanh(c_t)
        return h_t, c_t
```
~~~

Used in a single lstm model as follows:

~~~
```{python}
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.cell = LSTMCell(input_size, hidden_size)

    def __call__(self, x):
        B, T, _ = x.shape
        h = mx.zeros((B, self.hidden_size), dtype=mx.float32)
        c = mx.zeros((B, self.hidden_size), dtype=mx.float32)

        outputs = []
        for t in range(T):
            h, c = self.cell(x[:, t, :], (h, c))
            outputs.append(h)

        return mx.stack(outputs, axis=1), (h, c)

```
~~~

And finally the bidirectional lstm is set up along the text classification:

~~~
```{python}

class BiLSTM(nn.Module):
    def __init__(self, input_size, hidden_size):
~~~

~~~
        super().__init__()
        self.fwd = LSTM(input_size, hidden_size)
        self.bwd = LSTM(input_size, hidden_size)

    def __call__(self, x):
        B, T, _ = x.shape

        fwd_out, _ = self.fwd(x)

        rev_idx = mx.arange(T - 1, -1, -1, dtype=mx.int32)
        x_rev = mx.take(x, rev_idx, axis=1)

        bwd_out_rev, _ = self.bwd(x_rev)
        bwd_out = mx.take(bwd_out_rev, rev_idx, axis=1)

        return mx.concatenate([fwd_out, bwd_out], axis=-1)  # (B, T, 2H)


class BiLSTMTextClassifier(nn.Module):
    def __init__(self, embedding_dim, hidden_size, num_classes, dropout=0.1):
        super().__init__()
        self.bilstm = BiLSTM(embedding_dim, hidden_size)
        self.dropout = nn.Dropout(dropout)
        self.head = nn.Linear(2 * hidden_size, num_classes)

    def pool(self, seq_out, mask):
        mask_f = mask.astype(mx.float32)[..., None]
        summed = mx.sum(seq_out * mask_f, axis=1)
        denom = mx.maximum(mx.sum(mask_f, axis=1), 1e-6)
        return summed / denom

    def __call__(self, x, mask):
        # zero out padding before recurrence
        x = x * mask.astype(mx.float32)[..., None]
        seq_out = self.bilstm(x)
        pooled = self.pool(seq_out, mask)
        pooled = self.dropout(pooled)
        return self.head(pooled)


```
~~~

To use the framework to it’s full poterntial the model its is compiled

~~~
```{python}
@partial(mx.compile, inputs=state, outputs=state)
def step(Xb, yb, Mb):
~~~

~~~
    loss, grads = loss_and_grad(model, Xb, yb, Mb)
    optimizer.update(model, grads)
    return loss

```
~~~

The description of the loss function is omitted but can be found in the complete code in the repository

6 Naive Bayes Baseline

As a classical probabilistic benchmark, a Naive Bayes classifier is employed alongside the neural sequence model. To ensure comparability, all preprocessing steps prior to feature extraction—data cleaning, label normalization, and the exact train/validation/test split—are identical to those used for the BiLSTM classifier.

6.1 Model description

In contrast to the sequence-based neural architecture, the Naive Bayes model operates on a sparse bag-of-ngrams representation of the text. Each document x is mapped to a TF–IDF feature vector

$$\mathbf{f}(x) = (f_1, f_2, ..., f_d), \tag{9}$$

where each f_j reflects the TF–IDF weight of term j after vocabulary truncation and filtering. The vectorizer uses standard preprocessing settings for text classification:

- lowercasing
- English stop-word removal
- unigram and bigram features $(1, 2)$
- a maximum feature cap of $d = 50,000$
- a minimum document frequency of 2

The classifier itself is a Multinomial Naive Bayes model, which assumes conditional independence of features given a class label $y \in 0, 1$. Under this assumption, the likelihood of observing the feature vector $\mathbf{f}(x)$ given class k factorizes as

$$p(\mathbf{f}(x) \mid y = k); \propto; \prod_{j=1}^{d} \theta_{kj}^{,f_j(x)}, \tag{10}$$

where θ_{kj} are class-conditional feature probabilities estimated from the training set. Bayes' rule then yields the posterior

$$p(y = k \mid x); \propto; p(y = k) \prod_{j=1}^{d} \theta_{kj}^{,f_j(x)}, \tag{11}$$

and prediction is made via

$$\hat{y} = \arg\max_{k \in 0,1} p(y = k \mid x). \tag{12}$$

Although the multinomial model is derived for count data, it is widely used with TF–IDF features and remains a strong linear baseline in high-dimensional text classification.

7 Empirical Assessments

This section reports the empirical performance of the two main classifiers: the BiLSTM sequence model and the Naive Bayes baseline.

7.1 BiLSTM

7.1.a Headline results

On the held-out test partition, the BiLSTM achieves:

$$\text{Test loss} = 0.5229, \qquad \text{Test accuracy} = 0.7952.$$

{#ewq-bilstm-acc}

Validation- and test-set performance are closely aligned, suggesting stable generalization without pronounced overfitting.

7.1.b Class-wise metrics

Class	Precision	Recall	F1	Support
0	0.7621	0.6748	0.7158	12729
1	0.8121	0.8697	0.8399	20573
Accuracy	NA	NA	0.7952	33302
Macro avg	0.7871	0.7723	0.7779	33302
Weighted avg	0.7930	0.7952	0.7925	33302

Performance is systematically stronger for class 1, which is also the majority class in the data and therefore this behaviour is somewhat expected. The macro-averaged F_1 is slightly lower than the weighted average, reflecting the impact of class imbalance on aggregate metrics.

7.1.c Confusion matrix and derived rates

The test-set confusion matrix for the BiLSTM is:

$$\mathbf{C} = \begin{pmatrix} 8590 & 4139 \\ 2681 & 17892 \end{pmatrix}. \tag{13}$$

Treating class 1 as the positive class, the following quantities are obtained:

$$
\begin{aligned}
\text{TPR (recall }_1\text{)} &= \frac{17892}{17892 + 2681} = 0.8697, \\
\text{TNR (specificity)} &= \frac{8590}{8590 + 4139} = 0.6748, \\
\text{FPR} &= \frac{4139}{8590 + 4139} = 0.3252, \\
\text{FNR} &= \frac{2681}{17892 + 2681} = 0.1303, \\
\text{Balanced accuracy} &= \frac{0.8697 + 0.6748}{2} = 0.7723, \\
\text{MCC} &= 0.5592.
\end{aligned}
\tag{14}
$$

7.1.d Error profile

The confusion matrix indicates two main error patterns:

- **Class 0 → Class 1**: A comparatively large number of false positives (items from class 0 predicted as class 1) likely arises from shared lexical or stylistic features between the two parties, amplified by class imbalance which then leads to a bigger set of possible patterns are beeing labeled as class 1.
- **Class 1 → Class 0**: Fewer false negatives for class 1, which may correspond to more moderate or cross-partisan language that is less prototypical of the majority class.

The false-positive rate is roughly twice the false-negative rate, implying that the classifier tends to err toward predicting class 1. In applications with asymmetric error costs, this tendency could be counteracted by threshold adjustment, class-weighted loss functions, or post-hoc calibration.

The distribution of posts lengths does not have an impact on the error profile as throughout the dataset the post length distribution is similar across classes as shown in Section 3 .

7.2 Qualitative Analysis

To show the qualitative implications of the model a sample of posts is classified here.

7.2.a Single words

The model is given a list of the most comon english words and their predicted probabilities for each class. The results are visualized in Figure 2, which plots the margin (p_dem – p_repub) for each word against its rank in the sorted list. Points are colored by margin and sized by confidence (maximum predicted probability). The twenty most extreme words on either end of the spectrum are labeled.

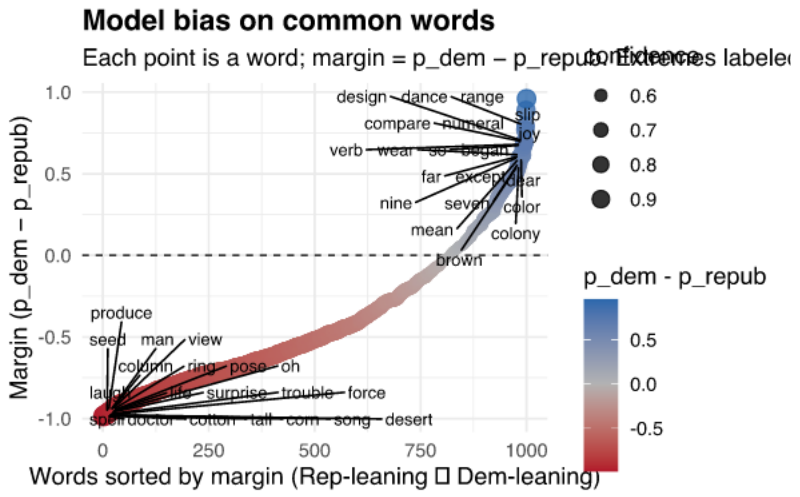


Figure 2: Model bias on common words

As Figure Figure 2 indicates, the classifier shows pronounced partisan leanings even when evaluated on isolated tokens.

A natural next step is to test whether this behavior is stable over time. In particular, one could examine whether applying the model to contemporary Republican and Democratic posts changes its performance, given well-documented shifts in political communication and issue framing over recent years.

7.3 Naive Bayes Baseline

7.3.a Headline results

As a bag-of-words baseline, the Naive Bayes classifier attains the following accuracy on the test set:

$$\text{Test accuracy} = 0.7843 \tag{15}$$

Given its simplicity and lack of contextual or sequential modeling, this level of performance is comparatively strong, but still below that of the BiLSTM.

7.3.b Class-wise metrics

Class	Precision	Recall	F1	Support
0	0.7698	0.6214	0.6877	12729
1	0.7907	0.8850	0.8352	20573
Accuracy	NA	NA	0.7843	33302
Macro avg	0.7803	0.7532	0.7615	33302
Weighted avg	0.7827	0.7843	0.7788	33302

As with the BiLSTM, performance is higher for class 1 than for class 0. Precision and recall for class 1 are clearly stronger, indicating a tendency to assign documents to the majority class and to capture its lexical cues more reliably.

7.3.c Confusion matrix and error profile

The corresponding test-set confusion matrix for the Naive Bayes classifier is

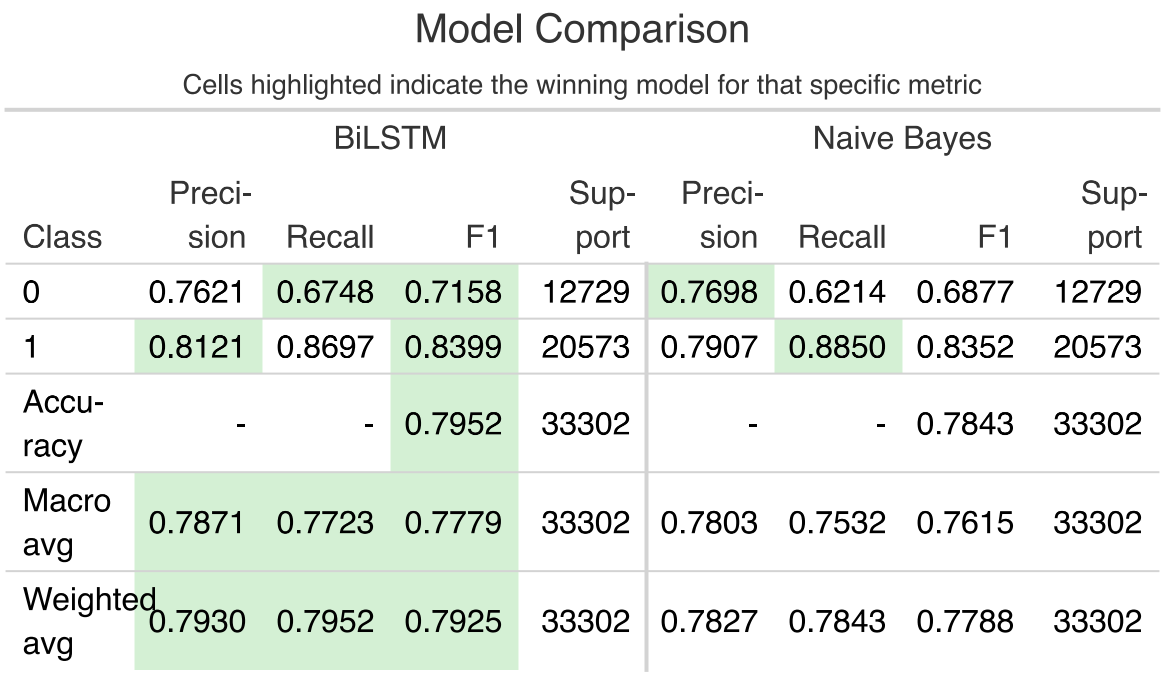
$$\begin{pmatrix} 9599 & 3171 \\ 2025 & 18666 \end{pmatrix}. \tag{16}$$

False positives for class 1 (3171 cases) are moderately more frequent than in the BiLSTM. Overall, the baseline captures strong lexical signals at low computational cost but offers less balanced performance across classes than the sequence model.

7.4 Comparative perspective

To assess whether the BiLSTM improves upon the Naive Bayes baseline in a systematic way, a side-by-side comparison of key metrics is provided in Table 1. The BiLSTM offers modest gains in overall accuracy and more favorable error balance across classes, while the Naive Bayes classifier remains a competitive and interpretable lexical baseline.

Table 1: Model Comparison



Model Comparison

Cells highlighted indicate the winning model for that specific metric

	BiLSTM				Naive Bayes			
Class	Precision	Recall	F1	Support	Precision	Recall	F1	Support
0	0.7621	0.6748	0.7158	12729	0.7698	0.6214	0.6877	12729
1	0.8121	0.8697	0.8399	20573	0.7907	0.8850	0.8352	20573
Accuracy	-	-	0.7952	33302	-	-	0.7843	33302
Macro avg	0.7871	0.7723	0.7779	33302	0.7803	0.7532	0.7615	33302
Weighted avg	0.7930	0.7952	0.7925	33302	0.7827	0.7843	0.7788	33302

8 Limitations and possible extensions

8.1 Limitations

The primary constraint of the current model is its reliance on frozen feature extraction. Because the Transformer $\mathcal{F}$ functions purely as a static input generator with stop-gradient operations, the embedding space is unable to adapt to the specific political vernacular of the congressional dataset during backpropagation. This lack of domain adaptation is compounded by precision limitations; the utilization of a 4-bit quantized Transformer and the subsequent casting of embeddings to `float16` introduces quantization noise that, while efficient, may discard subtle semantic distinctions found in full-precision representations. Furthermore, the architecture exhibits potential redundancy by stacking a BiLSTM on top of a powerful contextual encoder. Since the MiniLM Transformer already captures long-range dependencies via self-attention, the addition of recurrence introduces a sequential bottleneck without necessarily increasing modeling capacity significantly. Finally, the masked mean pooling strategy treats all valid tokens as equally important, a simplification that risks diluting strong, localized discriminative signals within longer posts.

8.2 Extensions

To address these limitations, several architectural and training modifications could be implemented. A significant performance boost could be achieved through end-to-end fine-tuning, where the Transformer layers are unfrozen to allow gradients to flow through $\mathcal{F}$. To manage the memory overhead of this approach on Apple Silicon, techniques such as Low-Rank Adaptation (LoRA) could be employed to fine-tune the encoder efficiently. Alternatively, efficiency could be prioritized by moving to a Transformer-only architecture, removing the BiLSTM layer entirely and projecting the Transformer's `[CLS]` token directly to the classification head; this would simplify the compute graph and eliminate the sequential dependency. Regarding feature aggregation, replacing mean pooling with a learnable attention mechanism would allow the model to weigh informative keywords more heavily than neutral connective text. Finally, while the current model truncates inputs at 128 tokens, implementing a sliding window approach or utilizing the full 512-token capacity of MiniLM would better capture tail-end context in lengthy statements.

9 Conclusion

The implemented pipeline offers a computationally efficient neural baseline for binary party attribution in political texts. By combining subword token IDs with frozen embeddings and a BiLSTM encoder, the system captures sequential patterns and achieves solid test performance. Evaluation reveals a mild bias toward predicting the majority class, visible in an elevated false-positive rate for class 0.

10 External sources used

- list of most common english words
- Dataset
- MLX-Embeddings repo
- MLX Repositories
- MLX Implementation of xLSTM
- Pytorch implementaino of LSTM

Bibliography