IMDB Movie Review Sentiment Analysis Using a Simple RNN

Summary

This report documents a complete, end-to-end sentiment prediction pipeline built on the **IMDB movie reviews dataset**. The goal is to predict whether a given movie review has a **positive** or **negative** sentiment.

The process followed:

- Explore the dataset structure and label distribution
- Convert token IDs back to words to inspect preprocessing
- Pad sequences to fixed length for model compatibility
- Train a SimpleRNN model with embeddings and early stopping
- Evaluate with confusion matrix, precision/recall/F1, and ROC–AUC
- Implement a single-review scoring pipeline for live predictions

The final model achieves **88% accuracy** and an **AUC of ~0.95** on the hold-out test set. Both precision and recall are balanced at ~0.88, showing strong classification.

1. Business Problem Formulation

Objective. Predict the probability that a given review is positive (binary classification).

Decision. Classify as positive when predicted probability ≥ decision threshold (default: 0.50).

Why SimpleRNN?

- Reviews are sequential data, where word order matters.
- SimpleRNN captures dependencies between words, unlike bag-of-words.
- Serves as a lightweight baseline before moving to more complex architectures like LSTM/GRU/Transformers.

2. Data Overview

Source: tf.keras.datasets.imdb (pre-tokenized integer sequences).
Size: 25,000 reviews for training, 25,000 for testing.

Labels: 1 = Positive, 0 = Negative.
Vocabulary limit: Top 10,000 most frequent word

The IMDB dataset assigns integer IDs to words based on frequency rank in the training corpus. Setting num_words=10000 means we only keep the 10,000 most frequent words, replacing all others with <UNK>.

Without this cap, the vocabulary could be ~88,000 words. A large vocabulary means more parameters in the embedding layer, meaning more memory, more training time.

Also, rare words may only appear once or twice, they add noise rather than signal. The most frequent words cover the majority of the text's meaning.

3. Exploratory Data Analysis (EDA)

3.1 Sentiment Distribution

Question: Is the dataset balanced between positive and negative reviews?
Findings:

- Train: Negative = 12,500 (50%), Positive = 12,500 (50%)
- Test: Negative = 12,500 (50%), Positive = 12,500 (50%)
  Implication: No class imbalance handling is required.

3.2 Decoding Tokenized Reviews

Decoding lets us translate the numeric representation of reviews back into human-readable text.

In the IMDB dataset, each review is stored as a sequence of integer IDs, where each ID corresponds to a word in the dataset's vocabulary.

1. Special Tokens

The first 4 integer IDs are reserved for structural markers, not actual words:

ID	Token	Purpose
0	<PAD>	Padding token to make all reviews the same length
1	<START>	Marks the start of a review

2	<UNK>	Unknown or out-of-vocabulary word
3	<UNUSED>	Reserved for future use

2. Vocabulary Mapping

word_index (from imdb.get_word_index()) maps words to their raw frequency (10,000) ranks, without accounting for the reserved IDs.

Example (truncated):

```
word_index = {
    "the": 1,
    "and": 2,
    "a": 3,
    "this": 14,
    "film": 22,
    "was": 16,
    ...
}
```

When Keras encodes reviews into X_train and X_test, it adds an offset of +3 to every actual word ID so that IDs 0–3 remain available for the special tokens above.

- "the" → stored as 4 (1 + 3)
- "this" → stored as 17 (14 + 3)
- "film" → stored as 25 (22 + 3)
- "was" → stored as 19 (16 + 3)

Why we need to shift when building reverse_index:

If we invert word_index directly, "the" would be linked to ID 1, but in X_train, "the" is 4.

So we do:

reverse_index = {value + 3: key for key, value in word_index.items()}

reverse_index[0] = "<PAD>"

reverse_index[1] = "<START>"

reverse_index[2] = "<UNK>"

reverse_index[3] = "<UNUSED>"

This ensures the lookup (word_index) matches the IDs actually used in the encoded reviews.

3. Example

Encoded review:

[1, 17, 25, 19, 46, 533, 976, 1625]

Decoding process:

Token ID	reverse_index lookup	Word Meaning
1	<START>	Start of review
17	"this"	this
25	"film"	film
19	"was"	was
46	"just"	just

533	"brilliant"	brilliant
976	"casting"	casting
1625	"location"	location

Decoded review:

<START> this film was just brilliant casting location

4. Data Preparation

4.1 Padding Sequences

X_train = sequence.pad_sequences(X_train, maxlen=500)

X_test = sequence.pad_sequences(X_test, maxlen=500)

Neural networks (like SimpleRNN, LSTM, GRU) require all inputs in a batch to have the same shape so they can be represented as tensors. Movie reviews in the IMDB dataset vary in length, like some have only a few words, while others can have thousands.

Padding makes every review exactly maxlen tokens long. Shorter reviews are padded with <PAD> (0) at the start by default. Longer reviews are truncated to fit maxlen.

Original (3 tokens):

[1, 56, 78]

Padded to length 10:

[0, 0, 0, 0, 0, 0, 0, 1, 56, 78]

In this project, maxlen=500, so the same principle applies, but every sequence is exactly 500 tokens long.

In the IMDB dataset, ~95% of reviews are 500 tokens or fewer after tokenization. Choosing 500 means it keeps almost all the original content for most reviews, without cutting too much.

RNNs process sequences step by step, so doubling maxlen doubles computation. A much longer length (e.g 2,000) would capture every word but slow training. Very long reviews often

contain repetitive or less relevant text, which can introduce noise. Truncating after 500 keeps the most informative first part of the review.

5. Model Design

5.1 Architecture

Embedding(10000, 128, input_length=500)

→ SimpleRNN(128, activation='relu')

→ Dense(1, activation='sigmoid')

Embedding Layer

Embedding converts each word (represented as an integer ID from the IMDB vocabulary) into a dense 128-dimensional vector.

Neural networks cannot directly process raw integer IDs because:

- IDs are labels, not numerical features.
- Example: "film" might be ID 22 but that number has no mathematical relationship to "movie" (ID 87).

How it works:

- **Input:** Sequence of integers (after padding)
  Example encoded review:
  [1, 14, 22, 16, 43, 530, 973]
  → <START> this film was just brilliant casting
- **Output:** Sequence of vectors, each with 128 learned features.
  Example (first 3 words only):
  <START> → [0.10, -0.21, ..., 0.34] (128 values)

  this → [0.12, -0.45, ..., 0.77] (128 values)

  film → [-0.05, 0.18, ..., -0.22](128 values)

- The embedding weights adjust so semantically similar words have small cosine distances.
  Example: "good" and "great" might be close in vector space.

Key Parameters:

- 10000 → Only the top 10,000 most frequent words are kept. Rare words are replaced with <UNK> (ID 2).
- 128 → Embedding dimension (number of features per word vector).
- input_length=500 → Every review is padded or truncated to exactly 500 tokens for uniformity.

SimpleRNN Layer (128 units, ReLU activation)

- RNN processes the sequence word-by-word, remembering past context via a hidden state.

Example

Using the first 3 words of the review:

1. **Time Step 1**:
    - Input: <START> vector → Hidden state $h_1 = \text{ReLU}(W_x * x_1 + W\square * h_0 + b)$
    - h_0 starts as a zero vector.
2. **Time Step 2**:
    - Input: "this" vector → $h_2 = \text{ReLU}(W_x * x_2 + W\square * h_1 + b)$
3. **Time Step 3**:
    - Input: "film" vector → $h_3 = \text{ReLU}(W_x * x_3 + W\square * h_2 + b)$

Here:

- $x\square$ = embedding vector at time t
- $h\square$ = hidden state at time t (size 128)
- W_x = weights for the current input
- $W\square$ = weights for the previous hidden state
- b = bias term

After the last word (time step 500), the final hidden state contains the "summary" of the review's meaning.

ReLU (f(x) = max(0, x)), avoids the vanishing gradient problem more than sigmoid/tanh in hidden layers, allowing better learning and not loosing weights updation on the early worlds.

SimpleRNN however has issues with very long sequences because information from earlier steps fades as you move forward (known as the short-term memory problem). For long-term dependencies, LSTM or GRU layers perform better.

Dense Output Layer (Sigmoid activation)

- Takes the final hidden state h_final from the RNN.
    - Computes:
      y_hat = sigmoid(W_out * h_final + b_out)
- Output: A probability between 0 and 1.
- Interpretation:
    - ≥ 0.5 → Predict Positive review
    - < 0.5 → Predict Negative review

Full Example

Sample review:

"This film was just brilliant"

1. **Encoded IDs:** [1, 14, 22, 16, 43, 530]
2. **Embedding Layer Output:** 6 × 128 matrix of dense vectors.
3. **RNN Steps:**
    - t=1: <START> updates h_1
    - t=2: "this" updates h_2
    - t=3: "film" updates h_3 … until "brilliant" updates h_6
4. **Dense Output:**
    - Sigmoid outputs 0.91 → Predict Positive (91% confidence).

5.2 Optimization & Training

Optimizer – Adam (learning rate = 0.0001)

- Combines the Momentum (smooths updates using past gradients) and RMSProp (scales learning rate per parameter).
- Adapts learning rate for each weight, making training more stable.
- Small learning rate 0.0001 → prevents overshooting the optimal weights

Loss – Binary Cross-Entropy

- Formula:
  Loss = -[y·log(ŷ) + (1-y)·log(1-ŷ)]
    - y = true label (0 or 1)
    - ŷ = predicted probability from Sigmoid
- Good for binary classification where output is a probability.

EarlyStopping (patience=5, monitor='val_loss', restore_best_weights=True)

- Monitors val_loss after each epoch.
- If no improvement for 5 consecutive epochs, training stops early.
- Restores the model to the weights from the best epoch.
- Prevents overfitting by stopping before the model starts memorising noise.
- Example:
    - Epoch 8 → lowest val_loss = 0.35
    - Epochs 9–13 → val_loss stays ≥ 0.35
    - Training stops at epoch 13, weights from epoch 8 are restored.

6. Training Output

Validation Accuracy: Stabilized around 0.87
Loss Curves: Training and validation loss decreased smoothly → stable optimization.

7. Evaluation on Hold-out Set (threshold = 0.50)

Confusion Matrix:

	Predicted Negative	Predicted Positive
Actual Negative	11028	1472
Actual Positive	1530	10970

Metrics:

- Accuracy: 0.8799
- Precision (Positive): 0.8817
- Recall (Positive): 0.8776
- F1-score: 0.8796
- AUC (using probabilities): 0.9503

Insights:

- Balanced precision/recall shows the model is equally good at catching positives and avoiding false alarms.
- High AUC means strong ranking ability: if pick a random positive and a random negative review, the model will assign the positive a higher score ~95% of the time.
- ~12% of each class misclassified → common in text sentiment tasks due to sarcasm/mixed wording/or vocabulary limits.

8. Production-Ready Scoring Pipeline (Single Review)

Steps implemented:

1. Load saved model (simple_rnn_imdb.h5).
2. Convert input review to lowercase and split into words.
3. Map each word to IMDB word_index ID, OOV → <UNK> (2), then +3 shift.
4. Pad/truncate to length 500.
5. Predict sentiment probability with model.
6. Classify using threshold.

Example:

predict_sentiment("This was a great movie!")

```
# Output: Positive, 0.92


predict_sentiment("I DID NOT LIKE THE MOVIE!")

# Output: Negative, 0.08
```

9. Recommendations & Next Steps

9.1 Improve Long-Term Context

- Replace SimpleRNN with LSTM or GRU to remember longer dependencies.

9.2 Regularization

- Add Dropout to RNN to reduce overfitting.

9.4 Threshold Tuning

- Adjust threshold for different precision/recall trade-offs depending on business needs.

9.5 Text Preprocessing

- Handle contractions (“don’t” → “do not”), strip punctuation, normalize case.