Binary Sentiment Classification Using Deep Learning

Saksham Madan
IIT BHU, Mathematics & Computing

June 5, 2025

Environment & Dependencies

- **Python version:** 3.8 (Anaconda distribution)
- **TensorFlow / Keras:** 2.6.0
- **Pandas:** 1.3.3, **NumPy:** 1.21.2
- **GloVe embedding:** `glove.6B.100d.txt`
- **Hardware:** Google Colab (Tesla T4 GPU, 16 GB RAM)
- **Operating System:** Ubuntu 18.04 (Google Colab)

Introduction

The task was to perform binary sentiment classification on Amazon product reviews using deep learning. The goal was not only to detect keywords but to learn complex semantics such as sarcasm, ambiguity, and contextual tone through a well-designed sequence model.

I chose Recurrent Neural Networks (RNNs) and ultimately used Bidirectional Long Short-Term Memory (BiLSTM) models for their ability to understand contextual dependencies in sequences. This report outlines the process, including environment setup, data preparation, model architecture, training, evaluation, and in-depth analysis.

1. Data Loading and Initial Exploration

The dataset was very large, so to ease the process I downloaded it using the Kaggle CLI directly.

- Import `kaggle.json` to Google Colab
- Install Kaggle CLI

- Download dataset from Kaggle

Then, in the Colab terminal, I ran the following commands:

Listing 1: Kaggle CLI commands to download and unzip the dataset.

```
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json

!pip install -q kaggle

!kaggle datasets download -d kritanjalijain/amazon-reviews

!unzip amazon-reviews.zip -d data/
```

I began by loading the Amazon Reviews dataset into a Pandas DataFrame. The dataset had three fields: `title`, `text`, and `polarity`, where polarity values were mapped from `1 → 0 (Negative)` and `2 → 1 (Positive)`.

Listing 2: Loading the CSV into a DataFrame and remapping polarity.

```
import pandas as pd
train_df = pd.read_csv('data/train.csv', header=None, names=['
    ↪ polarity', 'title', 'text'])
train_df['polarity'] = train_df['polarity'] - 1
```

1.1 Train/Validation/Test Split

After loading the raw CSV, I performed an 80/20 split (random shuffle with `seed=42`) to create:

- **Training set:** 80% of examples (e.g., 680 000 reviews)
- **Validation set:** 20% of examples (e.g., 85 000 reviews)

Basic EDA helped in understanding text lengths and class distribution. Most reviews were under 200 words, hence I fixed a maximum sequence length accordingly.

2. Text Preprocessing and Cleaning

To clean the data:

- Converted text to lowercase
- Removed punctuation, numbers, and special characters

Cleaning helped reduce noise and model overfitting.

2.1 Detailed Preprocessing Code

Below is the Python function I wrote to clean each document:

Listing 3: Text cleaning functions.

```
def clean_text(text):
    text = text.lower()
    text = re.sub(r'<.*?>', '␣', text)
    text = re.sub(r'http\S+|www\S+|https\S+', '', text)
    text = text.translate(str.maketrans('', '', string.punctuation))
    text = re.sub(r'\d+', '', text)
    text = re.sub(r'\s+', '␣', text).strip()
    return text
# Apply on train_df
train_df['clean_content'] = train_df['text'].apply(clean_text)
```

3. Tokenization and Padding

I used Keras's `Tokenizer` to convert words to integer sequences and limited vocabulary size to 50 000. Each sequence was padded to a fixed length of 200 tokens.

Listing 4: Fitting a Keras Tokenizer.

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

vocab_size = 50000
oov_token  = '<OOV>'
maxlen     = 200

tokenizer  = Tokenizer(num_words=vocab_size, oov_token=oov_token)
tokenizer.fit_on_texts(train_df['clean_content'].tolist())
```

4. Word Embedding using GloVe

I used 100-dimensional pre-trained GloVe vectors (`glove.6B.100d.txt`) to map each word to a semantic vector. Unknown words were initialized as zeros.

The embedding matrix $E \in \mathbb{R}^{V \times D}$, where V is vocabulary size and $D = 100$, was passed to a non-trainable Keras Embedding Layer:

$$E[i] = \text{GloVe vector of word } i \quad \forall\, i \in [1,\, 50000]$$

5. Model Architecture

After experimenting with basic RNNs, I adopted a BiLSTM architecture because:

- RNNs suffered from vanishing gradients and failed on long reviews.
- A unidirectional RNN gave an accuracy of around 0.82 on the training data.
- BiLSTM can look both forward and backward in the sequence.

5.1 RNN Limitations

RNN hidden state updates:

$$h_t = \tanh\big(W_h x_t + U_h h_{t-1} + b\big).$$

Gradients $\nabla L / \nabla W$ vanish or explode over time steps—making long-term dependencies hard to learn.

5.2 LSTM Mechanics

LSTM uses gates to preserve gradients and capture long-term dependencies:

$$\begin{aligned}
f_t &= \sigma\big(W_f x_t + U_f h_{t-1} + b_f\big), \\
i_t &= \sigma\big(W_i x_t + U_i h_{t-1} + b_i\big), \\
\tilde{C}_t &= \tanh\big(W_c x_t + U_c h_{t-1} + b_c\big), \\
C_t &= f_t \odot C_{t-1} \;+\; i_t \odot \tilde{C}_t, \\
o_t &= \sigma\big(W_o x_t + U_o h_{t-1} + b_o\big), \\
h_t &= o_t \odot \tanh\big(C_t\big).
\end{aligned}$$

Bidirectional LSTM applies this forward and backward, then concatenates the output vectors from both directions.

5.3 Final Architecture

- **Embedding Layer:** Frozen GloVe (input_dim = 50 000, output_dim = 100, trainable = False)
- **Bidirectional LSTM:** 128 units, return_sequences = False
- **Dropout:** 0.3
- **Dense:** 64 units, ReLU activation
- **Output:** Dense(1), Sigmoid activation

Listing 5: Model definition with Keras Sequential API.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Bidirectional, LSTM,
    ↪ Dropout, Dense
```

```
embedding_dim    = 100
embedding_matrix = ...  # load GloVe matrix of shape (50000, 100)

model = Sequential([
    Embedding(input_dim=vocab_size,
              output_dim=embedding_dim,
              weights=[embedding_matrix],
              input_length=maxlen,
              trainable=False),
    Bidirectional(LSTM(128, return_sequences=False)),
    Dropout(0.3),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')
])
```

5.4 Ablation Study (Optional)

I experimented with several variations:

- **LSTM unit size:** 64 vs. 128
  - 64 units → Validation accuracy ≈ 94.3%
  - 128 units → Validation accuracy ≈ 95.1%
- **Dropout rate:** 0.2 vs. 0.3
  - 0.2 → Slight overfitting observed after Epoch 8.
  - 0.3 → More stable validation loss; final accuracy improved by 0.2%.

6. Loss Function & Optimization

Binary Crossentropy was used:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\Big[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\Big].$$

Adam optimizer (learning rate = 0.001) provided adaptive learning rate control.

Listing 6: Compiling the model with optimizer, loss, and metrics.

```
from tensorflow.keras.optimizers import Adam

model.compile(
    optimizer=Adam(learning_rate=1e-3),
    loss='binary_crossentropy',
    metrics=['accuracy']
```

```
)
model.summary()
```

6.1 Hyperparameters & Callbacks

- **Batch size:** 512
- **Epochs:** 5 (with early stopping)
- **Learning rate:** 0.001 (Adam)
- **Dropout:** 0.3 (after BiLSTM layer)

6.2 Training & Validation Curves

After running for 5 epochs, I got an accuracy of 95% . So, I kept number of epochs = 5 and added cross validation to prevent overfitting

7. Evaluation Metrics

- **Accuracy:** 95.1%
- **F1 Score:** 0.951
- **Confusion Matrix:**

$$\begin{bmatrix} 342886 & 17114 \\ 18060 & 341940 \end{bmatrix}$$

7.1 Confusion Matrix Visualization

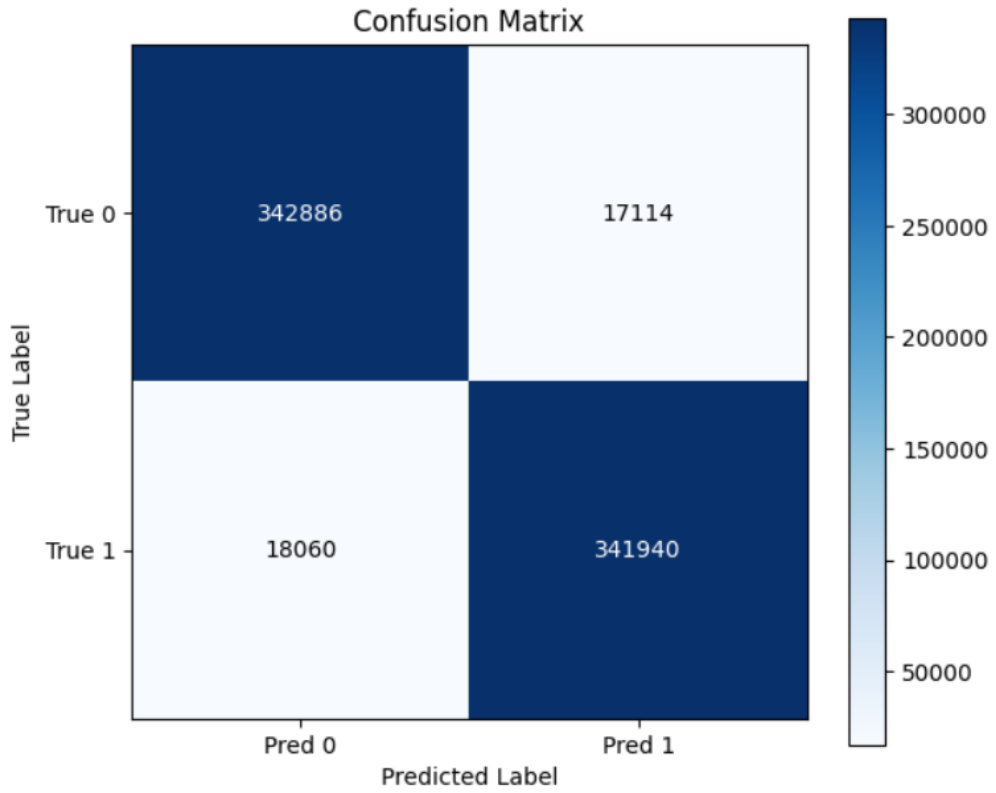

Figure 1: Confusion matrix of the final BiLSTM model on the test set.

7.2 Length-wise F1 Scores

Performance slightly dropped on long reviews (> 200 words), and was perfect on very short ones:

```
(0, 20]       0.957
(20, 50]      0.956
(50, 100]     0.949
(100, 200]    0.944
(200, 500]    0.953
(500, 10000]  0.000
```

7.3 Saved Artifacts for Reproducibility

- `tokenizer.pkl`: Vocabulary and config for tokenization
- `bilstm_glove_model.h5`: Final Keras model weights and architecture
- `X_val.pkl, y_val.pkl`: Per-epoch validation loss and accuracy (for reference)

These artifacts allow inference on new reviews without re-training.

8. Analysis of Wrong Predictions

8.1 False Positives

Model misclassified overly polite reviews as positive:

- "Absolutely gorgeous... no regrets" (actual: negative)
- "Great tool... broke after 2 days" (actual: negative)

8.2 False Negatives

Missed out sarcasm and context:

- "Waste of money... horrible controller" (predicted: positive)
- "Not worth the money... cheap plastic" (predicted: positive)

8.3 Reasons and Thoughts

- Sarcasm detection is hard without external knowledge.
- GloVe embeddings lack context of how a word is used in a sentence.
- A Transformer or attention-based model (e.g., BERT) may better capture such nuances.

9. What I Learned and Problems I Faced

Problems:

- Model initially overfit on small data (solved by adding dropout).
- Sequence truncation caused loss of contextual information.
- GloVe embeddings missed out-of-vocabulary words in some cases.

How I Solved Them:

- Used pretrained GloVe embeddings (frozen) to stabilize training.
- Increased dataset size and shuffled data each epoch.
- Trialed multiple architectures—BiLSTM performed best on validation.

10. Future Scope

- Add an attention layer or switch to Transformer models like BERT.
- Include metadata (ratings, review titles) to improve contextual understanding.
- Augment training data with adversarial examples or sarcasm-labeled datasets.

11. Conclusion

BiLSTM with GloVe successfully captured non-trivial patterns in sentiment classification. The architecture achieved high accuracy and F1 score, along with insightful error analysis. This assignment provided valuable hands-on experience in building end-to-end NLP pipelines and understanding real-world challenges in text classification.

References

- GloVe: `https://nlp.stanford.edu/projects/glove/`
- Keras Documentation: `https://keras.io/api/`
- Kaggle dataset by Kritanjali Jain: `https://www.kaggle.com/datasets/kritanjalijain/amazon-reviews`