# Report: Language Modeling and Classification with PyTorch

Natural Language Processing with Deep Learning Task 2
Almog Nimni
319090593

## Introduction

The project is divided into two primary parts:

1. **Task 1**: Building, training, and evaluating a Recurrent Neural Network (RNN) based Language Model on the IMDB movie review dataset.
2. **Task 2**: Conducting two experiments in text classification to predict sentiment (positive/negative).
   - **Experiment A**: Using the trained language model from Task 1 as a frozen "feature extractor" to train a classifier on a small subset (20%) of the data.
   - **Experiment B**: Building a new classifier using pre-trained Word2Vec embeddings, trained on a larger portion (80%) of the data.

This report will walk through each step, explaining the methodology, presenting the results, and concluding with a comparative analysis of the different approaches.

## Task 1: Language Modeling

The goal of this task was to build a model that can predict the next word in a sequence.

### Step 1: Exploratory Data Analysis (EDA) & Preprocessing

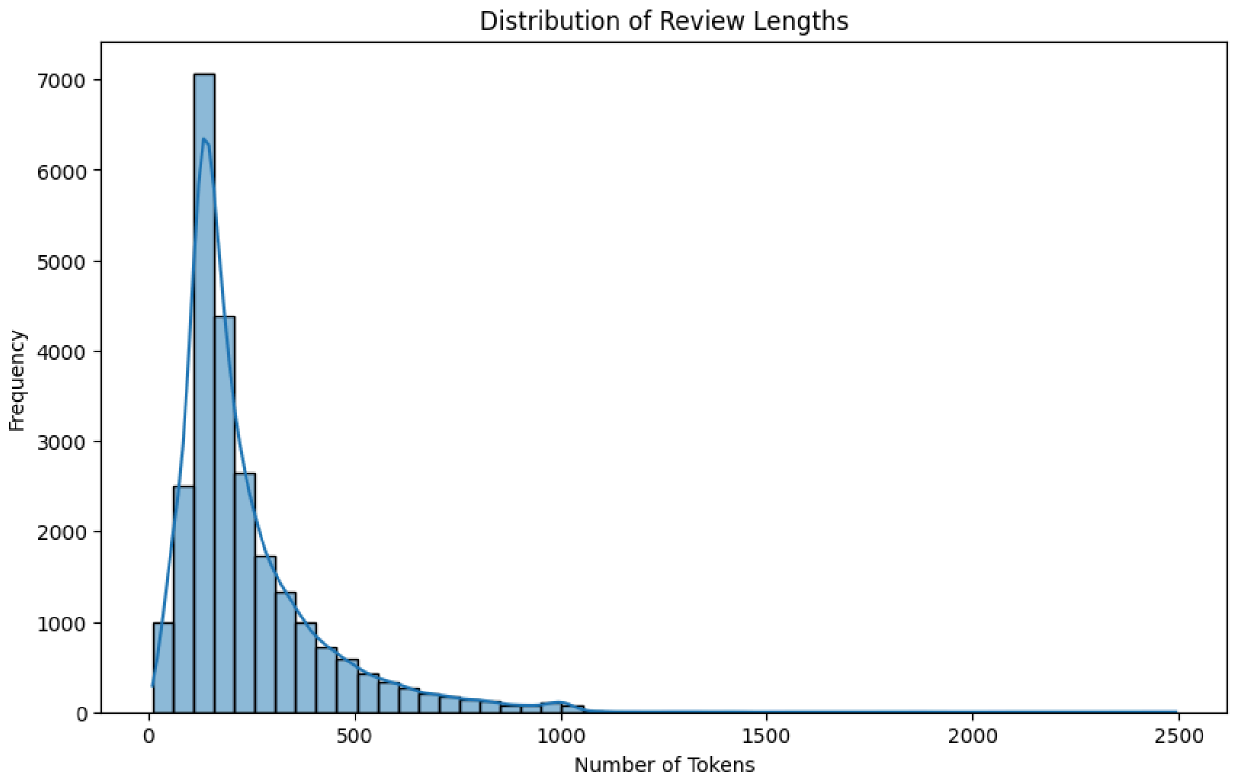
The first step was to understand the IMDB dataset. The dataset was loaded in the code itself using the Hugging Face “datasets” library.

Data Splitting:
To properly evaluate our model, the data was organized into three sets:

- **Training Set**: A 90% split of the original training data, used for training the model.
- **Validation Set**: A 10% split of the original training data, used to monitor performance during training and prevent overfitting.
- **Test Set (25,000 reviews)**: The original, untouched test set, reserved for the final evaluation of the model.

Data Analysis:
To understand the data behavior, I performed two key analyses:

1. Review Length Distribution: I tokenized the reviews and plotted a histogram of their lengths. The analysis showed a wide distribution, but most reviews were concentrated in the 100-300 token range. This insight was crucial for deciding on a max_len hyperparameter later to truncate long reviews, which prevents memory crashes and speeds up training.



2. **Word Frequency**: I counted the frequency of all tokens in the training set (results in the notebook). As expected, the most common words were "stopwords" like the, a, and is. For language modeling, these words are essential for learning sentence structure, so they were **not** removed.

## Step 2: Vocabulary and Data Objects

To feed text into a neural network, it must be converted into a numerical format. This was achieved through a data pipeline consisting of a Vocabulary, a PyTorch Dataset, and a DataLoader.

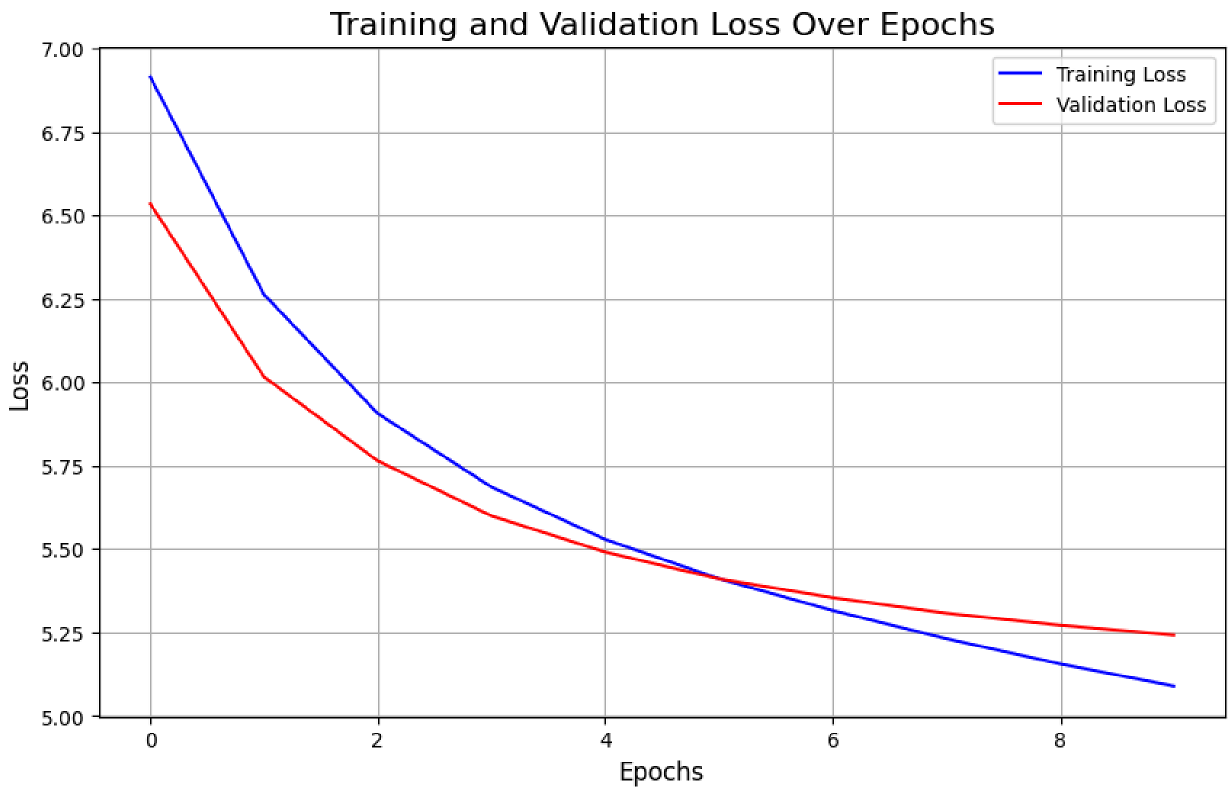
- **Vocabulary**: A Vocabulary class was created to map each unique word (token) to a unique integer index. To make the vocabulary more robust and manageable, it was built using only the training set, and words with a frequency of less than 5 were discarded and mapped to a special <unk> (unknown) token. A <pad> token was also added for batching. This resulted in a vocabulary of **29,111 words**.
- **Dataset**: A custom LanguageModelDataset class was implemented. Its key responsibility is to take a single review, tokenize it, use the vocabulary to convert it into a tensor of numerical indices, and then create the input-target pair required for language modeling. For a sequence of words [w1, w2, w3, ..., wn], the input to the model is [w1, w2, ..., wn-1] and the target is [w2, w3, ..., wn].
- **DataLoader**: Finally, the Dataset was wrapped in PyTorch DataLoaders for the training and validation sets. A custom collate_fn was used to pad all sequences in a batch to the same length, ensuring uniform tensor shapes for model processing.

## Step 3: Model Definition and Training

Model Architecture:
An RNN-based model was defined with the following layers:

1. **Embedding Layer**: Converts token indices into dense vectors of dimension 128, allowing the model to capture word meanings.
2. **LSTM Layer**: A two-layer Long Short-Term Memory (LSTM) network with a hidden dimension of 256. The LSTM processes the sequence of word vectors and captures long-range dependencies.
3. **Linear Layer**: A fully connected layer that maps the LSTM's output back to the vocabulary size, producing a probability distribution for the next word.

Training:
The model was trained for 10 epochs using the Adam optimizer and the Cross-Entropy Loss function, which is standard for multi-class classification problems like next-word prediction. The loss function was configured to ignore the padding token index, so it wouldn't impact the gradient calculations.
The training and validation loss were tracked at each epoch. The resulting graph shows a clear decrease in loss for both sets, indicating that the model was successfully learning. The validation loss consistently stayed close to the training loss, suggesting the model was not significantly overfitting.



**Step 4: Evaluation**

The model was evaluated using the standard metric for evaluating a language model is **Perplexity**, which is simply e^(loss). Perplexity measures how "confused" a model is by a sequence of text, lower perplexity is better.

The final trained model was evaluated on the unseen test set.

**Results:**

- **Test Loss:** 5.24
- **Test Perplexity:** 189.23

A perplexity of ~189 means that, on average, the model is as uncertain about the next word as if it were choosing randomly from 189 different words. Given a vocabulary size of over 29,000, this result is significantly better than random and demonstrates that the model successfully learned the underlying patterns of the English language in the IMDB dataset. The model and vocabulary were saved for use in Task 2.

## Task 2: Text Classification

### Experiment A: Pre-trained LM as a Backbone

This experiment tested the hypothesis that a language model trained on a specific domain (like movie reviews) can serve as a feature extractor for a classification task, even with limited labeled data.

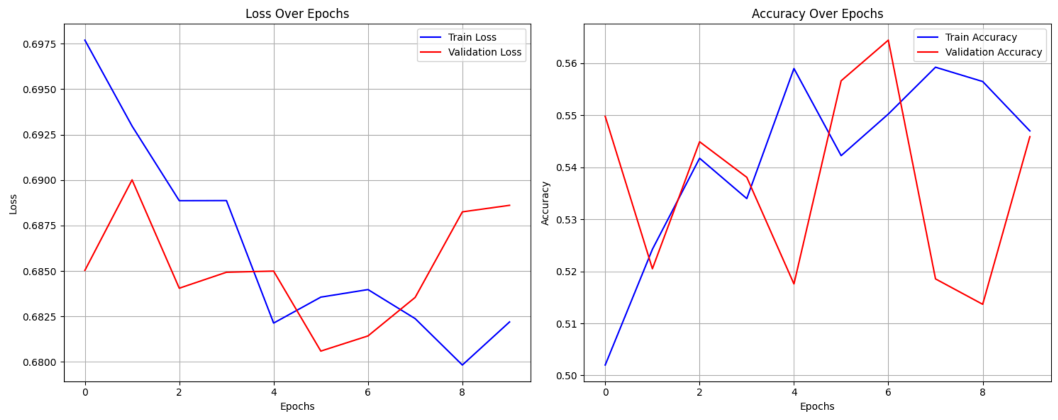
1. Data Preparation:
A small, stratified subset of 5,000 reviews (20% of the training set) was created. This subset was then split into a 4,000-sample training set and a 1,000-sample validation set. This was done to simulate a low-data scenario.

2. Model Architecture:
I built a SentimentClassifier model, consisting of:

- **Backbone**: The language model from Task 1. Its weights were **frozen** (requires_grad=False) to ensure that its learned language knowledge was preserved.
- **Sentence Encoder**: The output of the language model's LSTM for an entire review is a sequence of vectors. To get a single vector representing the whole sentence, I took the final hidden state from the LSTM's last layer.
- **Classifier Head**: A small, trainable Multi-Layer Perceptron (MLP) was added on top. It takes the sentence vector from the backbone and learns to map it to a single sentiment prediction (0 for negative, 1 for positive).

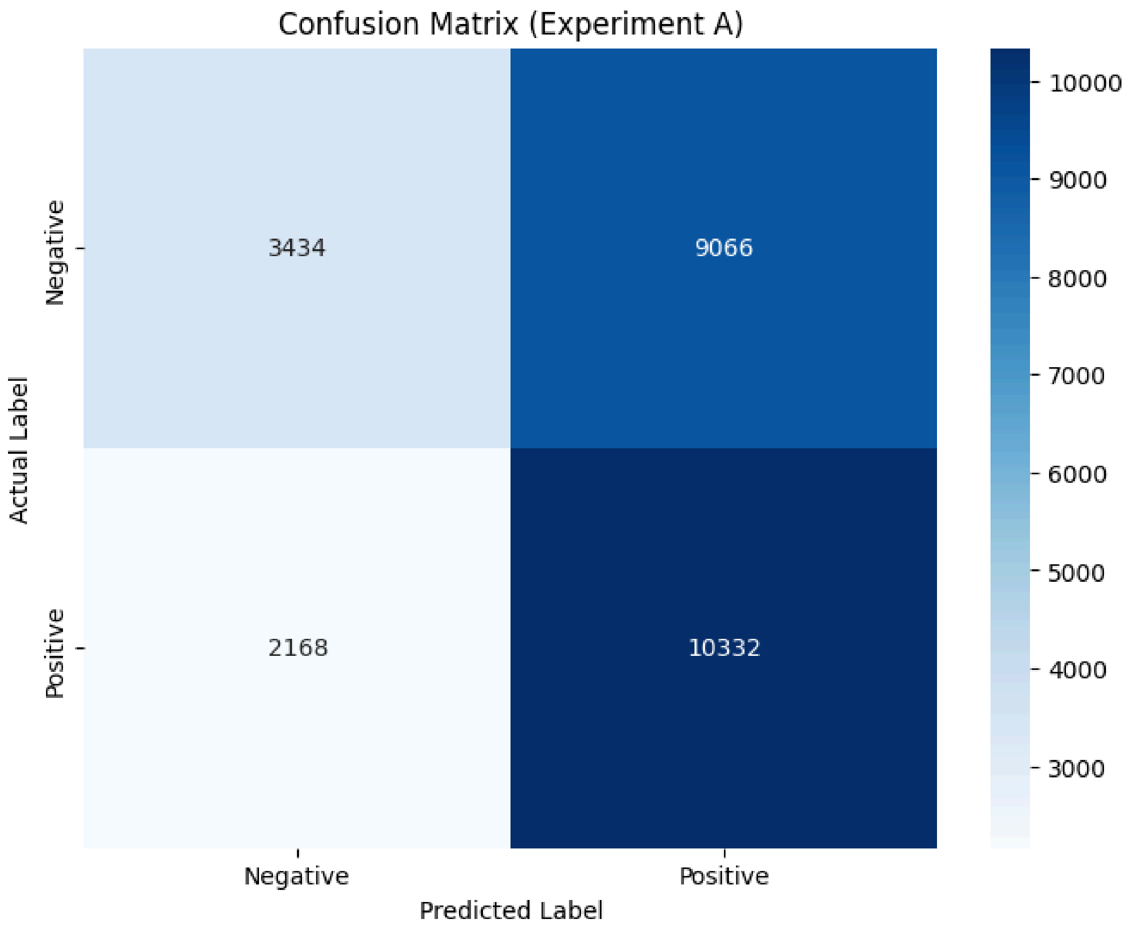
3. Training & Evaluation:

The model was trained for 10 epochs, but only the weights of the new classifier head were updated. The Adam optimizer and BCEWithLogitsLoss (Binary Cross-Entropy), which is suitable for binary classification, were used.



**Results (Experiment A):**

- **Test Accuracy:** 84.10%
- **Test F1-Score:** 0.855
- Confusion Matrix:



Error Analysis:
By examining the misclassified reviews, a few patterns emerged. The model struggled with:

- **Sarcasm and Nuance**: Reviews that used positive words to convey a negative sentiment (e.g., "This movie was 'brilliantly' bad").
- **Mixed Sentiment**: Reviews that praised one aspect (like acting) but heavily criticized another (like the plot).
- **Neutral or Plot-Summary Reviews**: Reviews that described the plot without strong emotional language were often difficult to classify correctly.

**Experiment B: Word2Vec + From-Scratch RNN**

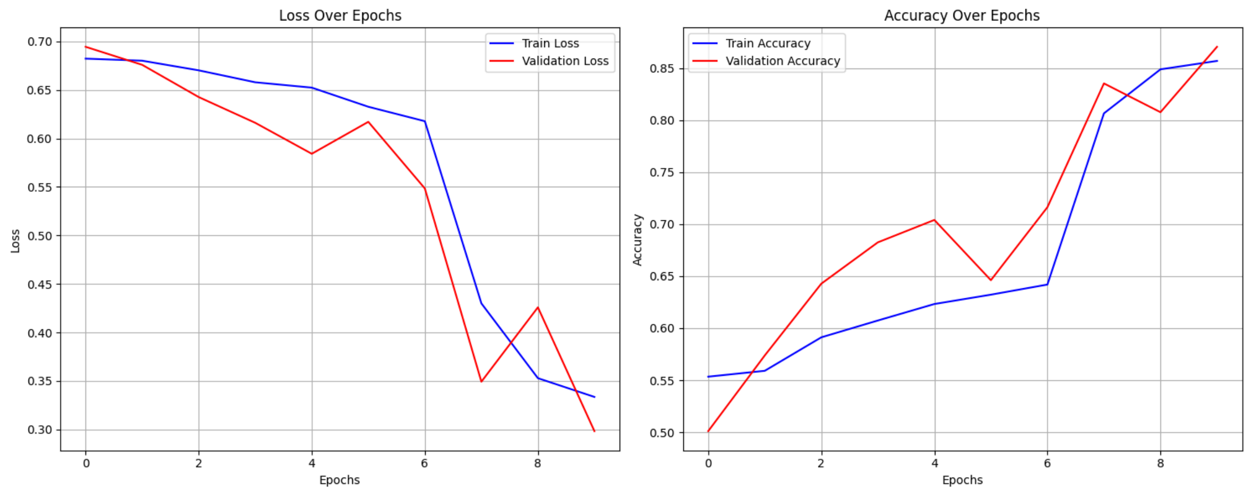
This experiment used a more traditional transfer learning approach: leveraging pre-trained word embeddings.

1. Data Preparation & Embeddings:
For this experiment, the model was trained on a larger dataset of 20,000 reviews (80% of the training data). A new vocabulary was built from this larger set.
Google's pre-trained word2vec-google-news-300 model was loaded using the gensim library. An embedding matrix was then created to bridge our dataset's vocabulary with Word2Vec's. For each word in our vocabulary, its corresponding 300-dimensional vector was retrieved from Word2Vec and placed in our matrix. Words not found in Word2Vec were left as zero vectors.

2. Model Architecture:
I created sentimentClassifierW2V model which was built with the following layers:

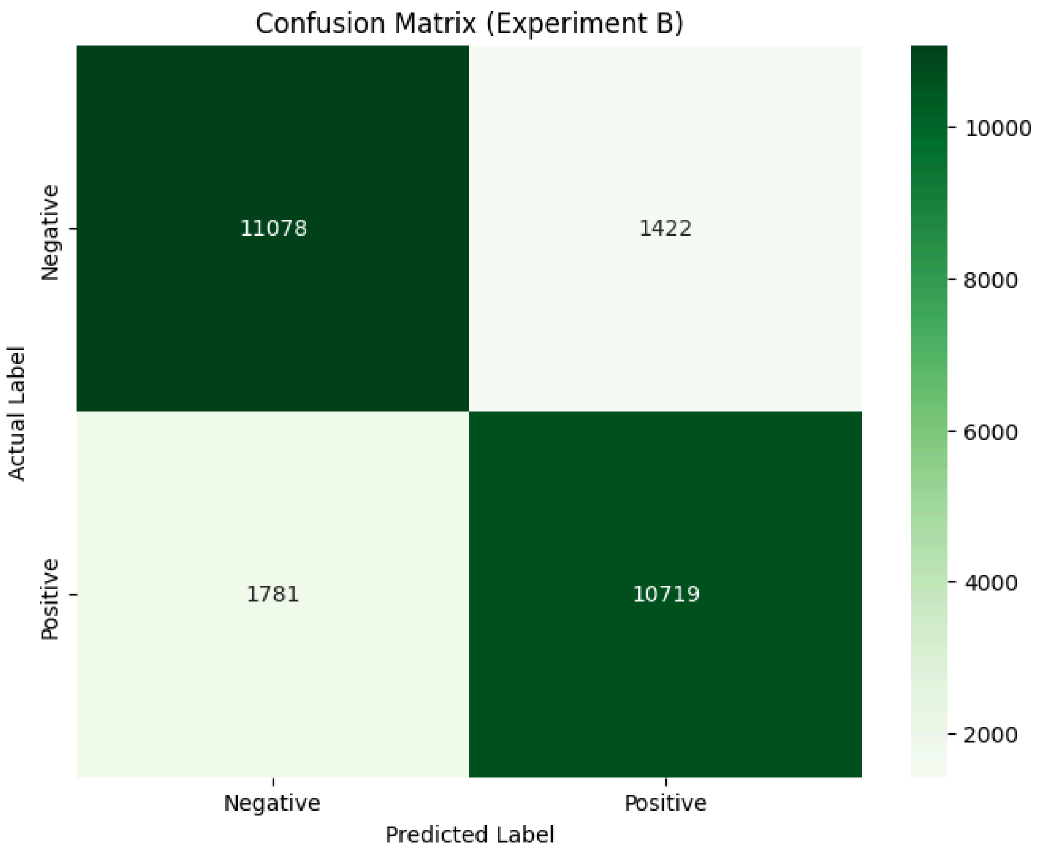
- **Embedding Layer**: Initialized with the custom Word2Vec embedding matrix and **frozen** to preserve the pre-trained word knowledge.
- **Bidirectional LSTM**: A two-layer bidirectional LSTM was used to capture context from both forward and backward directions in the text. This layer was trained from scratch.
- **Classifier Head**: A trainable linear layer for the final classification.

3. Training & Evaluation:
The model was trained and evaluated using the same process as Experiment A.



**Results (Experiment B):**

- **Test Accuracy:** 84.04%
- **Test F1-Score:** 0.852
- Confusion Matrix:



Error Analysis:
The errors made by this model were very similar to those in Experiment A, primarily involving reviews with complex sentiment, sarcasm, and neutral language.

## Comparison between the models and Conclusions

| **Metric** | **Experiment A (LM Backbone)** | **Experiment B (Word2Vec)** |
|---|---|---|
| Training Data Size | 4,000 (20%) | 20,000 (80%) |
| Test Accuracy | **84.10%** | 84.04% |
| Test F1-Score | **0.855** | 0.852 |

**Key Conclusions:**

1. **Effectiveness of a Domain-Specific LM**: The most striking result is that **Experiment A achieved slightly better performance while using only one-fifth of the training data**. This strongly suggests that a language model pre-trained on a domain-specific corpus (IMDB reviews) is an incredibly effective feature extractor for tasks within that same domain. It learned not just word meanings, but also the context, phrasing, and style specific to movie reviews, which was more valuable than the general-purpose word knowledge from Word2Vec.
2. **Data Efficiency**: The LM backbone approach is far more data-efficient. Achieving ~84% accuracy with only 4,000 samples is a testament to the power of transfer learning from a fine-tuned model. The Word2Vec approach required 5 times more data to achieve a similar result.
3. **Trade-offs**:
    - **LM Backbone (A)**: Requires a significant upfront cost to pre-train the language model. However, once trained, it can be used for downstream tasks with very little labeled data, making it ideal for low-resource scenarios.
    - **Word2Vec (B)**: This approach is faster to set up since the embeddings are readily available. However, it requires a larger labeled dataset for the main task model (the LSTM) to learn the contextual patterns from scratch.

In conclusion, both experiments were successful in building effective sentiment classifiers. However, the results clearly demonstrate the superior data efficiency and performance of using a custom language model, pre-trained on the target domain, as a feature extractor. This highlights a key trend in modern NLP: fine-tuning large, contextual models is often more powerful than using static word embeddings alone.