Sentiment Analysis

Using Neural Networks

Javier Lopez

D213: Advanced Data Analytics

Table of Contents

Part I: Research Question

A1. Research Question

How does sentiment expressed in Amazon product reviews correlate with product ratings, and how can we use this information to predict future product success?

A2. Objectives

Our research question seeks to understand the relationship between the sentiment expressed in Amazon product reviews and the corresponding product ratings. More specifically, it aims to use this relationship to predict the future success of a product based on the sentiments expressed in its reviews. We've identified the following objectives to guide our data analysis:

- **Sentiment Analysis**
  We aim to develop a robust mechanism to extract and quantify the sentiment present in product reviews. The sentiment may be positive, negative, or neutral.
- **Correlation Assessment**
  Our next goal is to explore and understand the relationship between the expressed sentiment and the assigned product ratings. We want to ascertain if there's a clear correlation between the two, and the nature of this correlation, if it exists.
- **Predictive Modeling**
  Based on the correlation established, we aspire to build a predictive model that can forecast the likely success of a product, given a set of reviews. Success can be defined in several ways, such as future sales or ratings, which we'll need to specify based on available data and business needs.

- **Model Evaluation and Refinement**

  It's essential to rigorously evaluate our model's performance and make necessary refinements to enhance its predictive accuracy. We'll also be vigilant of common modeling pitfalls, such as overfitting.
- **Business Insights Generation**

  Lastly, we aim to translate our findings into actionable business insights that can guide decision-making related to product development, marketing strategy, and customer experience management.

A3. Neural Network Identification

One of the most popular types of neural networks for text classification tasks, such as sentiment analysis, is the Long Short-Term Memory network (LSTM), which is a type of Recurrent Neural Network (RNN). LSTMs are specifically designed to avoid the long-term dependency problem, which is the difficulty for an RNN in learning to connect past information to the present task, such as using past parts of a sequence to predict future parts. This makes them ideal for tasks involving sequential data, like text, where the order and context of words is important for understanding their meaning.

An LSTM network can take the sequence of words in a review and learn to predict the sentiment based on the learned contextual meaning of the words. It does this by maintaining a 'cell state', or a kind of working memory, that it can add or remove information to over time. This allows it to keep or discard information over long sequences, making it very effective for this kind of task. An LSTM network can be trained on a dataset of Amazon reviews and their associated sentiments to produce useful predictions for the sentiment of new reviews. The model would take as input a sequence of words in a review and output a prediction for the sentiment of that review.

Part II: Data Preparation

B1. Exploratory Data Analysis

B1.1. Presence of Unusual Characters

Navigating through our dataset, we found 123,958 observations with unusual characters such as emojis and non-English symbols. These characters aren't just noise; they add layers of meaning to reviews. Emojis, for instance, can provide cues about sentiment, a smile indicating positivity, a frown the opposite.

B1.2. Vocabulary Size

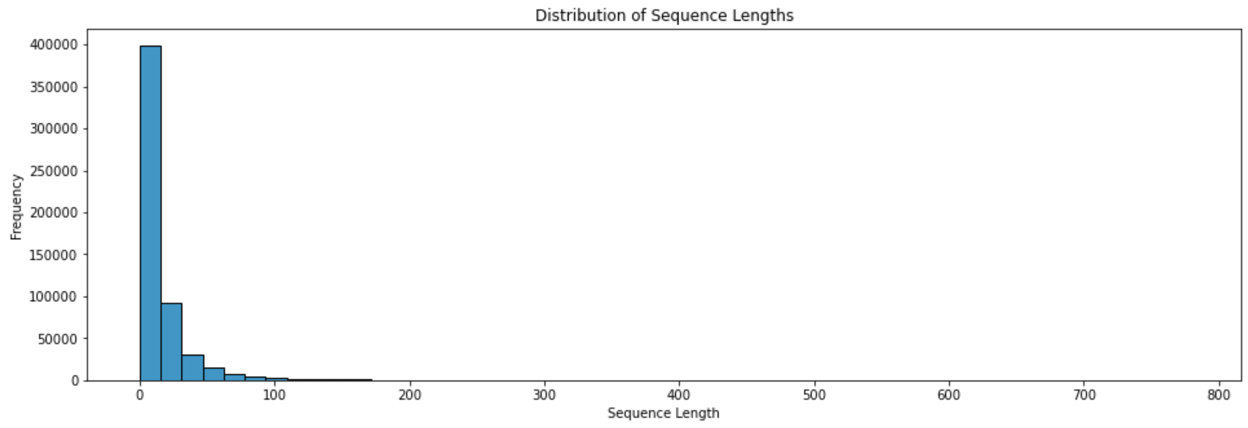
In our analysis, we've found that our word index comprises a total of 66,112 unique words. This substantial vocabulary size emphasizes the variety of language used in the Amazon reviews. It reflects the challenge our model faces in deciphering patterns, but also the potential depth of insight we can extract from these diverse expressions.

B1.3. Proposed Word Embedding Length

The decision to choose a specific word embedding length predominantly revolves around the compromise between computational efficiency and the volume of information the embedding can contain. An embedding length ranging between 100 and 300 dimensions is a common choice in numerous natural language processing tasks. Regarded as a hyperparameter of the model, the embedding length can be adjusted to achieve optimal performance. Given the intricacy of our text, a dimension of 200 emerges as a favored balance, harmonizing the complexity of the text with the computational resources we have allocated.

B1.4. Statistical Justification: Max. Sequence Length

The maximum sequence length is a crucial parameter that requires thoughtful setting when training a sequence model like an LSTM. One approach to determining this parameter is by examining the distribution of sequence lengths within the data. Inferences drawn from the summary statistics and the histogram plot reveal that 95% of all reviews possess a sequence length of 55 tokens. Thus, we plan to utilize 55 as the maximum sequence length, an approach that allows us to encapsulate most of the data without inducing overfitting to the model.



B2. Goals of the Tokenization Process

Tokenization is the technique of dividing text into pieces, commonly known as tokens. These tokens could range from sentences and words to subwords or even individual characters. The principal objectives we aim to achieve with tokenization include:

- Disassembling text into tokens, which subsequently become the fundamental units of text that the machine learning model can comprehend and learn from.
- Removing punctuation and unnecessary or special characters that do not offer beneficial information for our model. The keras Tokenization function is specifically tailored to manage this process seamlessly.

- Transforming words to lowercase to ensure that identical words in different cases are recognized as the same. The keras Tokenization function effortlessly facilitates this process.
- Removing HTML tags or other unique substrings that may be present due to web scraping. Again, the keras Tokenization function proves to be proficient at handling this task.

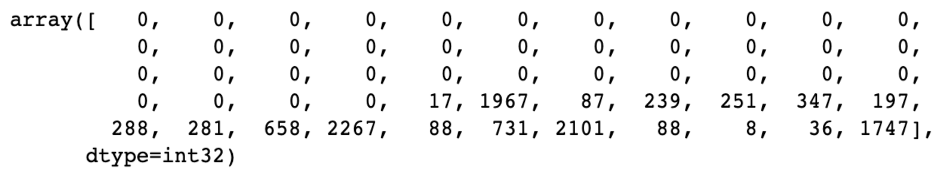
Normalization forms an additional process with the ultimate aim of converting the text into a standardized or regular format. This reduction of data complexity allows the machine learning model to more effectively learn from it. Our key objectives for normalization include:

- Implementing lemmatization, a process that reduces inflected or occasionally derived words to their root or stem form. For example, "runs", "running", "ran" are all different forms of the word "run".
- Removing stop words such as "is", "the", and "in", which typically lack substantial meaningful information."

B3. Sequence Padding

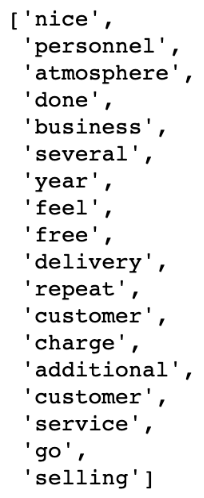
Padding is an indispensable preprocessing stage that ensures all sequences, or lists of tokens, within a dataset maintain a uniform length. This uniformity is a prerequisite as most machine learning models mandate their input to have a consistent shape. If sequences fall short of a certain length, padding steps in to fill the remaining spaces with 0, whereas sequences exceeding the length are generally truncated to match it.

We first decide on a sequence length. Choosing the sequence length often involves a trade-off: if it's too short, we might lose useful information; if it's too long, we might introduce too much noise from the extra padding. We then use the Keras pad_sequences function to apply padding to the sequences. The function takes a list of sequences (list of lists) and transforms them all into sequences of the same length. The image below shows a sample of a single padded sequence.



```
array([   0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,   17, 1967,   87,  239,  251,  347,  197,
        288,  281,  658, 2267,   88,  731, 2101,   88,    8,   36, 1747],
      dtype=int32)
```

The image below shows the original sequence before encoding and padding.



```
['nice',
 'personnel',
 'atmosphere',
 'done',
 'business',
 'several',
 'year',
 'feel',
 'free',
 'delivery',
 'repeat',
 'customer',
 'charge',
 'additional',
 'customer',
 'service',
 'go',
 'selling']
```

The decision between pre-padding and post-padding holds significant relevance when working with LSTM networks. Given their gating mechanisms, these networks may potentially "forget" information from earlier in the sequence. Consequently, it's generally more effective to pad at the sequence's beginning (pre-padding), as the network might not pay much attention to padding tokens.

B4. Categories of Sentiment and Activation Function

We will be doing multi-class classification (positive, neutral, and negative sentiment), where we will have 3 categories. To achieve this, we will encode the rating column (our target variable) as follows:

- Negative (rating 1 and 2), Encoded as [1,0,0]

- Neutral (rating 3), Encoded as [0,1,0]
- Positive (rating 4 and 5), Encoded as [0,0,1]

The final dense layer of the network will have 3 neurons, one for each class, and will use the softmax activation function, which outputs a probability distribution over the classes.

B5. Data Preprocessing Steps

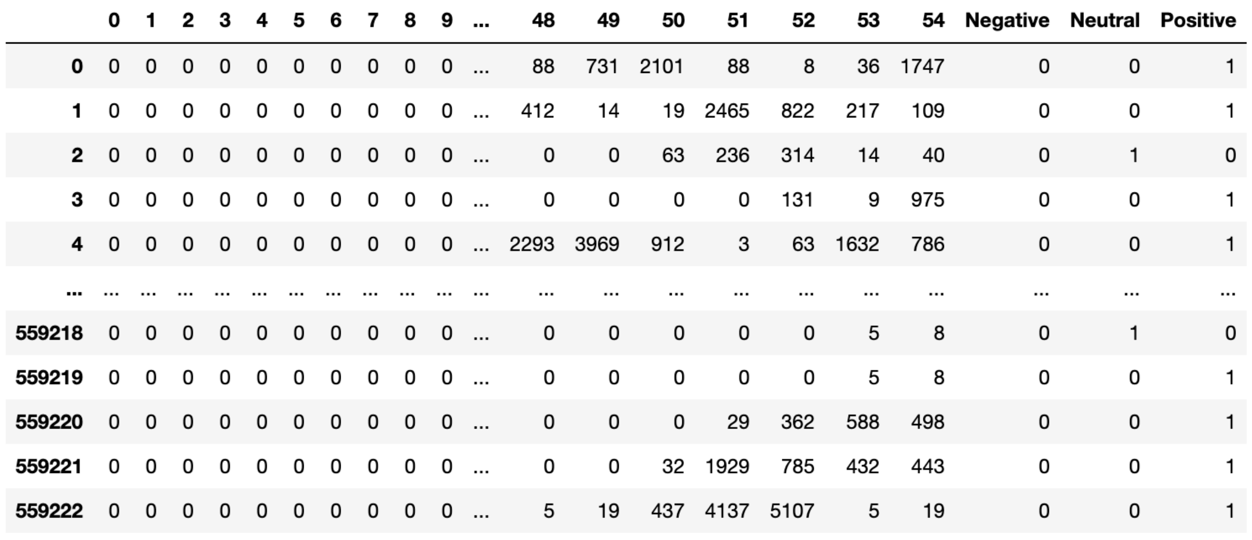
Data Cleaning: Our initial step involves addressing missing values, eliminating duplicates, and checking for data inconsistencies. In this scenario, it's essential to discard any reviews devoid of text.

Data Transformation: This phase involves reformatting our data into a format that's suitable for analysis. Specifically, it entails several sub-steps:

- **Text Normalization:** We'll transform all text into lowercase and eliminate punctuation.
- **Text Tokenization:** This involves deconstructing the reviews into individual words or tokens.
- **Removing Stop Words:** This step discards common words that carry minimal meaningful information, like 'the', 'a', 'in', etc.
- **Stemming/Lemmatization:** We'll condense words to their root or base form (for example, 'running' becomes 'run').
- **Encoding Text:** Here, we convert words into integers or vectors that serve as input for our neural network.
- **Encoding Labels:** This involves converting sentiment labels into one-hot vectors.
- **Padding Sequences:** We ensure all our text sequences possess uniform length by padding shorter ones with zeros.

Data Splitting: We'll divide the dataset into a training set, a validation set, and a test set. The training set is employed to train the model, the validation set aids in tuning parameters and adjusting model settings during training, and the test set evaluates the final model. Generally, a 70% split is allocated for training, 15% for validation, and 15% for testing.

B6. Prepared Dataset



	0	1	2	3	4	5	6	7	8	9	...	48	49	50	51	52	53	54	Negative	Neutral	Positive
0	0	0	0	0	0	0	0	0	0	0	...	88	731	2101	88	8	36	1747	0	0	1
1	0	0	0	0	0	0	0	0	0	0	...	412	14	19	2465	822	217	109	0	0	1
2	0	0	0	0	0	0	0	0	0	0	...	0	0	63	236	314	14	40	0	1	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	131	9	975	0	0	1
4	0	0	0	0	0	0	0	0	0	0	...	2293	3969	912	3	63	1632	786	0	0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
559218	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	5	8	0	1	0
559219	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	5	8	0	0	1
559220	0	0	0	0	0	0	0	0	0	0	...	0	0	0	29	362	588	498	0	0	1
559221	0	0	0	0	0	0	0	0	0	0	...	0	0	32	1929	785	432	443	0	0	1
559222	0	0	0	0	0	0	0	0	0	0	...	5	19	437	4137	5107	5	19	0	0	1

The image above reflects the prepared dataset in a DataFrame. The entirety of the prepared dataset is stored in *clean_reviews.json*.

Part III: Network Architecture

C1. Model Summary Output

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 embedding (Embedding)       (None, 55, 200)           13226800  
                                                                 
 lstm (LSTM)                 (None, 55, 128)           168448    
                                                                 
 dropout (Dropout)           (None, 55, 128)           0         
                                                                 
 lstm_1 (LSTM)               (None, 128)               131584    
                                                                 
 dense (Dense)               (None, 64)                8256      
                                                                 
 dense_1 (Dense)             (None, 3)                 195       
                                                                 
=================================================================
Total params: 13,535,283
Trainable params: 13,535,283
Non-trainable params: 0
_________________________________________________________________
```

C2. Network Layers and Parameters

Embedding Layer

Our embedding layer functions as a look-up table that transitions from integer indices (representing specific words) to dense vectors. With the bulk of our parameters (13,226,800), this layer is responsible for fabricating a 200-dimensional vector for each word in our vocabulary. The 'None' in the output shape signifies this layer's capability to accept sequences of any length.

LSTM Layers

Our architecture includes two LSTM layers with 128 units each. These layers are adept at managing sequential data, maintaining a memory of preceding inputs in the sequence. This feature is invaluable for sentiment analysis, where the sentence context is of crucial importance.

Our initial LSTM layer is configured to return sequences, meaning it outputs a sequence identical in length to the input, enabling information passage to the next LSTM layer.

Dropout Layer

To curb overfitting, we've incorporated a dropout layer. This layer randomly assigns zero to a fraction (50% in our case) of input units at each update during training time, a step that aids in preventing overfitting.

Dense Layers

In conclusion, we have two dense layers. The first houses 64 units and utilizes the 'ReLU' activation function. To further deter overfitting, this layer also carries L1 and L2 regularization, applying penalties to the layer's output and weights respectively. These penalties deter overly complex or extreme values, encouraging more generalizable data patterns. Our final dense layer is the output layer, featuring 3 units corresponding to the three sentiment classes we're predicting ('Negative', 'Neutral', 'Positive'). This layer leverages the 'softmax' activation function, delivering a probability distribution over the three classes.

Total Parameters

Altogether, our model possesses 13,535,283 parameters, all of which are trainable. The substantial quantity of parameters mainly originates from our embedding layer. Trainable parameters are those that the model assimilates from the training data, whereas non-trainable parameters remain static during training (our current model lacks any of these).

C3. Justification: Hyperparameters

Activation Functions

We opted for the 'ReLU' (Rectified Linear Unit) activation function for the penultimate layer as it assists in addressing the vanishing gradient issue, a common phenomenon in deep neural networks. It's computationally efficient and performs reliably in numerous scenarios. For our output layer, we deployed the 'softmax' function, a standard choice for multiclass classification tasks, as it outputs a probability distribution over our target classes ('Negative', 'Neutral', 'Positive').

Number of Nodes per Layer

We've set the LSTM layers to house 128 nodes each, with our dense layer containing 64. These quantities strike a balance between computational efficiency and the model's ability to learn intricate patterns. These figures are flexible, however, and may require adjustments depending on the task's performance.

Loss Function

We've employed 'categorical_crossentropy' as our loss function, a conventional choice for multiclass classification problems. It quantifies the dissimilarity between our predicted probability distribution and the actual distribution.

Optimizer

The 'adam' optimizer was chosen for its computational efficiency and modest memory demands. It's particularly suitable for problems involving substantial data and/or parameters. 'Adam' adjusts the learning rate for each weight individually, automating the tuning of the learning rate, which typically yields reliable results in practice.

Stopping Criteria

We have implemented early stopping, a regularization technique used to avoid overfitting when training a learner with an iterative method such as gradient descent. This method suspends the training process if the model's performance on the validation set does not improve for a certain number of epochs. In our case, this was an essential criterion to ensure our model does not overfit the training data.

Evaluation Metric

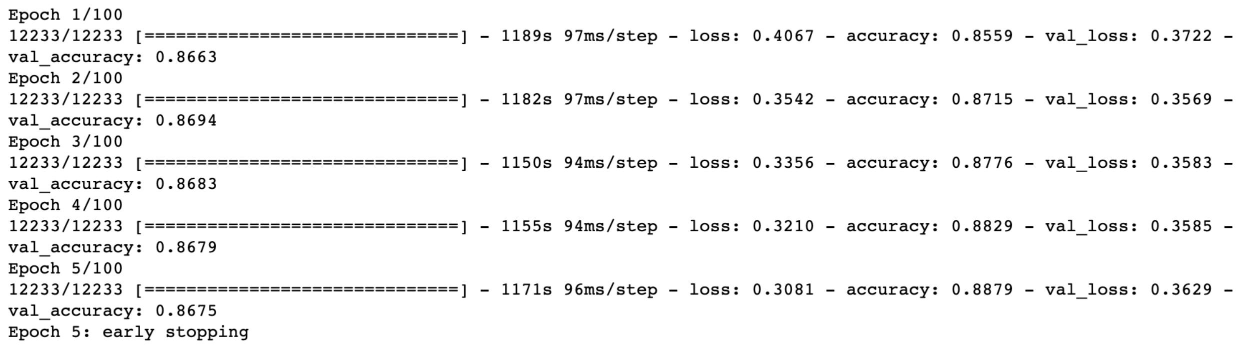
We chose 'Accuracy' as our evaluation metric for its simplicity and widespread use in classification tasks. It calculates the ratio of correct predictions to total predictions and can provide a more comprehensive understanding of our model's performance across diverse classes.

Part IV: Model Evaluation

D1. Impact of Using Stopping Criteria

Implementing early stopping as a criterion during our model's training process was a strategic decision to enhance model performance. By setting a defined number of epochs, we may run the risk of overfitting if the model trains for too long and begins to capture noise and inconsistencies in the training set. Alternatively, we could underfit the model if the training duration is inadequate, thereby missing out on vital patterns in the data.

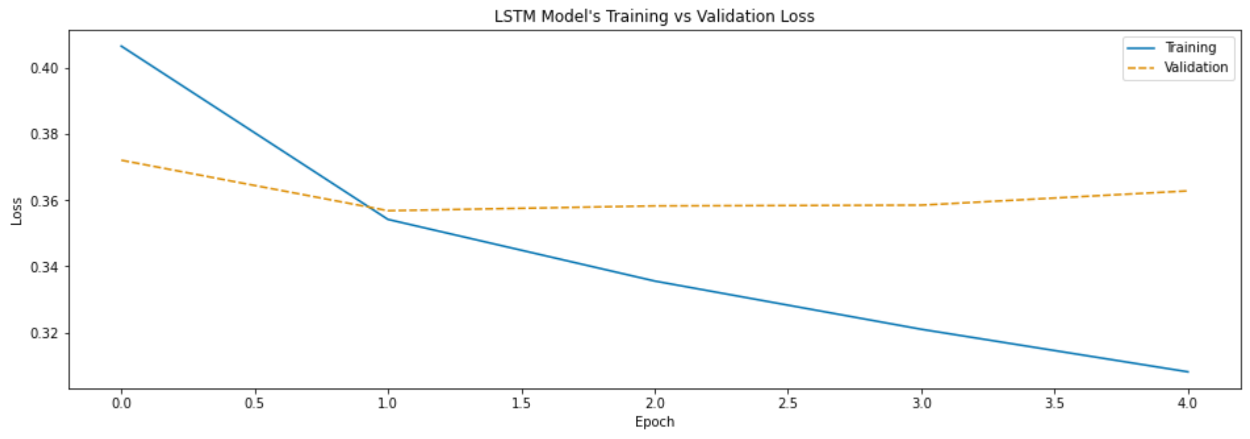
By implementing early stopping, we made our model more dynamic. The model was set to halt its training if it didn't register any improvement in the validation loss for a specific number of consecutive epochs. This approach makes the most of our training data by using it just as long as it is beneficial. Therefore, the model can autonomously decide when training ceases, ensuring an optimal balance between underfitting and overfitting. As seen from the final training epoch, early stopping efficiently halted the process once the validation loss ceased to improve. The image below shows the training process including the final training epoch.



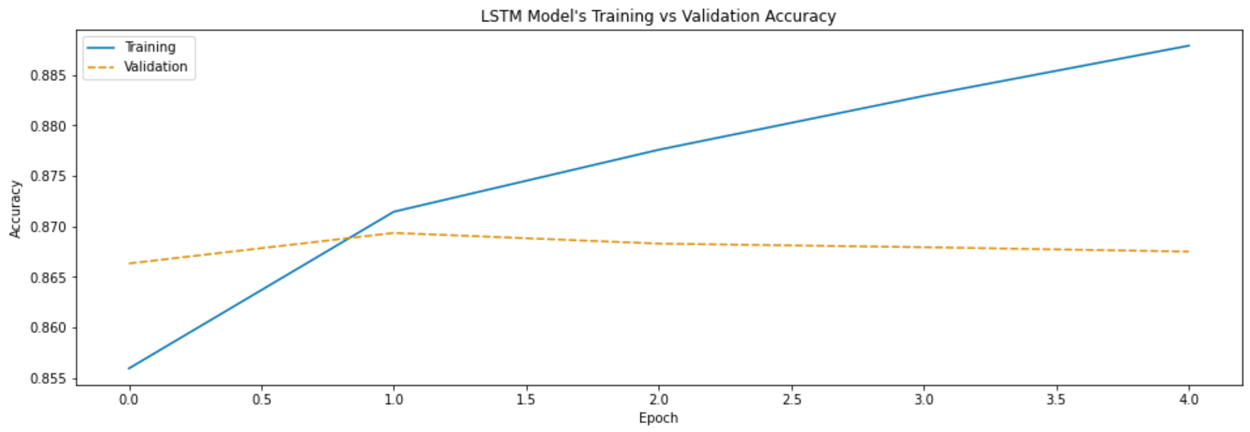
```
Epoch 1/100
12233/12233 [==============================] - 1189s 97ms/step - loss: 0.4067 - accuracy: 0.8559 - val_loss: 0.3722 -
val_accuracy: 0.8663
Epoch 2/100
12233/12233 [==============================] - 1182s 97ms/step - loss: 0.3542 - accuracy: 0.8715 - val_loss: 0.3569 -
val_accuracy: 0.8694
Epoch 3/100
12233/12233 [==============================] - 1150s 94ms/step - loss: 0.3356 - accuracy: 0.8776 - val_loss: 0.3583 -
val_accuracy: 0.8683
Epoch 4/100
12233/12233 [==============================] - 1155s 94ms/step - loss: 0.3210 - accuracy: 0.8829 - val_loss: 0.3585 -
val_accuracy: 0.8679
Epoch 5/100
12233/12233 [==============================] - 1171s 96ms/step - loss: 0.3081 - accuracy: 0.8879 - val_loss: 0.3629 -
val_accuracy: 0.8675
Epoch 5: early stopping
```

D2. Visualizations: Model Training Process

In the plots showcasing our model's training process, we get a vivid picture of how our model learned over time.



The loss plot reveals how the model consistently improved its ability to predict sentiment with each epoch, both on the training and validation sets. It's notable that the validation loss began to plateau and even slightly increase towards the end, indicating that the model was beginning to overfit.



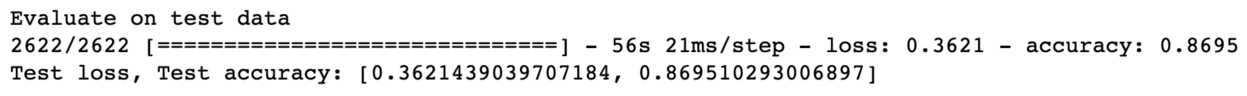
The accuracy plot complements the loss plot, illustrating the model's increasing proficiency in correctly predicting sentiment as training progressed.

D3. Model Fitness and Measures to Address Overfitting

The model, as a whole, demonstrates a good fit. It exhibits a gradual learning curve and manages to achieve a high accuracy on the training set. However, we noted signs of overfitting towards the end of the training process, where the validation loss started to increase, and validation accuracy plateaued. To counteract this, we implemented early stopping, which halted training once the model stopped improving on the validation set. Additionally, we incorporated dropout layers and L1 and L2 regularization within our model. These techniques randomly deactivate neurons and apply penalties to the model's weights, respectively, preventing the model from relying too much on any single pattern and encouraging it to find more generalizable underlying patterns.

D4. Predictive Accuracy

Our neural network has shown a commendable predictive accuracy of about 86.95% on unseen test data. This suggests that the model is proficient in its sentiment prediction capabilities when applied to real-world scenarios. However, it's important to bear in mind that even with this encouraging result, there's always room for refinement in the ever-evolving landscape of machine learning. The image below shows the results from evaluating the model on the unseen test data.



```
Evaluate on test data
2622/2622 [==============================] - 56s 21ms/step - loss: 0.3621 - accuracy: 0.8695
Test loss, Test accuracy: [0.3621439039707184, 0.869510293006897]
```

Part V: Summary and Recommendations

E. Code to Save the Trained Network

```
# Save the trained network
model.save('lstm_model.h5')  # create an HDF5 file 'lstm_model.h5'
```

To load the model from the saved file, one could use the following code:

```
# Load the model
loaded_model = load_model('lstm_model.h5')
```

F. Neural Network Functionality

Our neural network is structured to effectively handle the complex task of sentiment analysis on textual data. The architectural choices play an instrumental role in the model's efficacy, shaping how well it learns from our data, generalizes to unseen examples, and ultimately performs sentiment prediction. Starting off with the Embedding layer, this look-up table allows us to translate word indices into their corresponding vector representations. This layer is crucial as it enables our model to understand the semantic meaning embedded in each word and interpret how different words relate to one another. It's worth noting that the lion's share of our model's parameters resides in this layer due to its responsibility of crafting a 200-dimensional vector for each word in our vocabulary.

Moving on, the sequential nature of our textual data demanded the implementation of LSTM layers. With their ability to remember past inputs via internal memory, LSTM layers adeptly handle the dependencies between words in a sentence - an indispensable capability for sentiment analysis, as the sentiment often relies heavily on the context provided by the preceding words. The decision to use two LSTM layers augments the network's capacity to extract and interpret complex structures in the data. To mitigate overfitting, we strategically inserted a Dropout layer, which randomly sets a portion of its inputs to zero during each training step. This encourages the model to learn more robust, generalizable patterns in the data and lessens its tendency to over-rely on specific features.

The dense layers, or fully connected layers, act as the decision-making head of our network. The penultimate layer, with its 64 nodes, transforms the high-level features extracted by the preceding layers into a form suitable for final classification. By incorporating L1 and L2 regularization, we've ensured this layer doesn't produce overly complex or extreme values, further bolstering our model's generalizability. Our final layer houses three nodes corresponding to the three sentiment classes we're predicting. Utilizing the 'softmax' activation function, this layer effectively churns out a probability distribution over the classes, enabling us to identify the most probable sentiment for a given input review.

As a whole, our network's architecture is designed to handle the intricate task of sentiment analysis with grace. It intelligently combines the ability to understand semantic meanings, remember crucial context, generalize well, and make informed decisions based on the learned patterns. Each architectural choice has a tangible impact on our model's performance and has been meticulously considered to ensure our model's success in sentiment prediction.

G. Recommended Course of Action

Upon reflecting on the results and observations made during our exploration of sentiment analysis with LSTM networks, I would recommend the following course of action:

- **Experimentation with Network Architecture**

  Our current model demonstrates respectable performance, but there's always room for improvement. By varying the number of LSTM layers or their units, we could explore how these adjustments influence our model's accuracy. The addition of Bidirectional LSTMs, for instance, might enable the model to learn from both past and future context.

- **Hyperparameter Tuning**

  Experimenting with the model's hyperparameters, such as batch size, learning rate, or the dropout rate, could result in improved model performance. Grid Search or Random Search methods could be employed to systematically identify the optimal combination of these hyperparameters.

- **Improved Handling of Imbalanced Classes**

  Our model seemed to struggle slightly more with neutral reviews. Techniques like oversampling, undersampling, or generating synthetic samples could be explored to handle the class imbalance better, potentially enhancing the model's performance, particularly for underrepresented classes.

- **Enriched Preprocessing**

  Investigating other text preprocessing techniques like more sophisticated tokenization, employing n-grams, or better handling of negations could lead to more effective feature extraction, subsequently boosting our model's ability to discern sentiments.

- **Transfer Learning**

  Leveraging pre-trained word embeddings like Word2Vec or GloVe, or even more advanced language models like BERT, could enhance our model's understanding of semantic meanings, likely improving the sentiment prediction capabilities.

Through a combination of these tactics, we could refine our sentiment analysis model, ensuring it not only maintains its robust performance on the current dataset but also generalizes well to other similar tasks, bolstering its overall utility in real-world applications.

Part VI. Reporting

H. Industry-Relevant IDE File

The code for our model can be found in *Sentiment Analysis Using Neural Networks.ipynb*.

I. Sources

No additional sources were used in the creation of this project.

J. Citations

No in-text citations were used in the creation of this project.