D213 – Advanced Data Analytics

WGU M.S. Data Analytics

Lyssa Kline

April 10, 2025

A, Part 1: Research Question

A1, Research Question

The research question in scope for this analysis is “How accurately can customer sentiment (positive or negative) be predicted from short review sentences using a neural network model, and how can this analysis support Amazon in improving customer experience and decision-making?” This question leverages natural language processing (NLP) and neural networks to classify sentiment from real customer reviews; in this case, from amazon.com. Amazon relies heavily on customer reviews to influence purchasing decisions and guide product recommendations. In a real-world organization context, amazon could use this to automatically filter product reviews and flag negative feedback for quality control or seller alerts. Accurate sentiment analysis can help Amazon identify problematic products, enhance recommendation systems, and tailor marketing strategies.

A2, Analysis Goal and Objectives

The primary goals and objectives of this data analysis are to preprocess and clean the text data, build and train a neural network model, evaluate model performance, and generate actionable insights.

The first objective is to preprocess and clean the text data, this involves tokenizing the Amazon review sentences, removing irrelevant characters, and preparing the data for neural network input. Next, we can build and train the neural network model by using the labeled sentiment data to train a model that can learn the patterns in word usage associated with the sentiment. Once built, we can evaluate the model performance and assess the model using accuracy, precision, recall, and F1 score to determine how well it classifies sentiment in unseen Amazon review sentences. Lastly, we can start to generate actionable insights. This will provide Amazon with an automated approach to classify sentiment in real-time, enabling better product feedback loops, alerting on negative trends, and enhancing customer satisfaction efforts.

Each of these goals is directly correlated with the research question in scope and is achievable with the dataset provided, which includes 1000 Amazon sentences (500 positive and 500 negative).

A3, Type of Neural Network

A Recurrent Neural Network (RNN), specifically an LSTM (Long Short-Term Memory) network, is well-suited for this type of analysis. LSTMs are designed to handle sequential data, like sentences, where the order of words matters. They can capture long-term dependencies and contextual meaning in the text better than basic feedforward networks. Additionally, LSTMs are commonly used in NLP tasks like sentiment analysis, language modeling, and text generation.

This analysis was developed using a Sentiment Analysis in Google Colab with TensorFlow. The dataset is small enough that it is ideal for a first-time neural network project. TensorFlow offers high-level tools for building neural networks like LSTMs and handles binary sentiment labels well. Making this type of neural network a great fit for the resources provided.

B, Part 2: Data Preparation

B1, Exploratory Data Analysis

The data cleaning process for this analysis started with performing exploratory data analysis (EDA) to identify any discrepancies in the data. This is done to ensure the text data is clean and ready for modeling. The exploratory process consisted of identifying the presence of unusual characters, vocabulary size, proposed word embedding length, and the statistical justification for the chosen maximum sequence length.

Detecting unusual characters ensures the text is clean and consistent. A clean input ensures a better model focused on meaningful words rather than artifacts. To determine the presence of unusual characters, the data was searched for unusual characters using regular expressions targeting non-ASCII characters. The results showed that no unusual characters, emojis, or non-English symbols were present in the dataset, indicating that the Amazon review text was already well-formatted.

The size of the vocabulary tells you how complex your data is, how large your embedding matric will be and whether you need to filter out rare words. Too large a vocabulary can cause overfitting, increased memory usage, and longer training time. Too small might miss nuance. Determining the vocabulary size helps balance model complexity and efficiency by giving insights into the diversity of the words in the dataset. In this analysis, the review text was tokenized into words, converted to lowercase, and then counted the unique words. A secondary vocabulary size of 1879 was computed using TensorFlow's tokenizer, which excludes duplicates and special characters.

Word embedding is another important part of the EDA process. Word embeddings turn text into dense numerical vectors. An appropriate embedding dimension captures meaning without overloading the model. A word embedding dimension of 100 was used in the embedding layer of the neural network. A dimension of 100 is a common balance between performance and computational efficiency, particularly for small datasets. Its expressive enough to capture semantic relationship without overfitting.

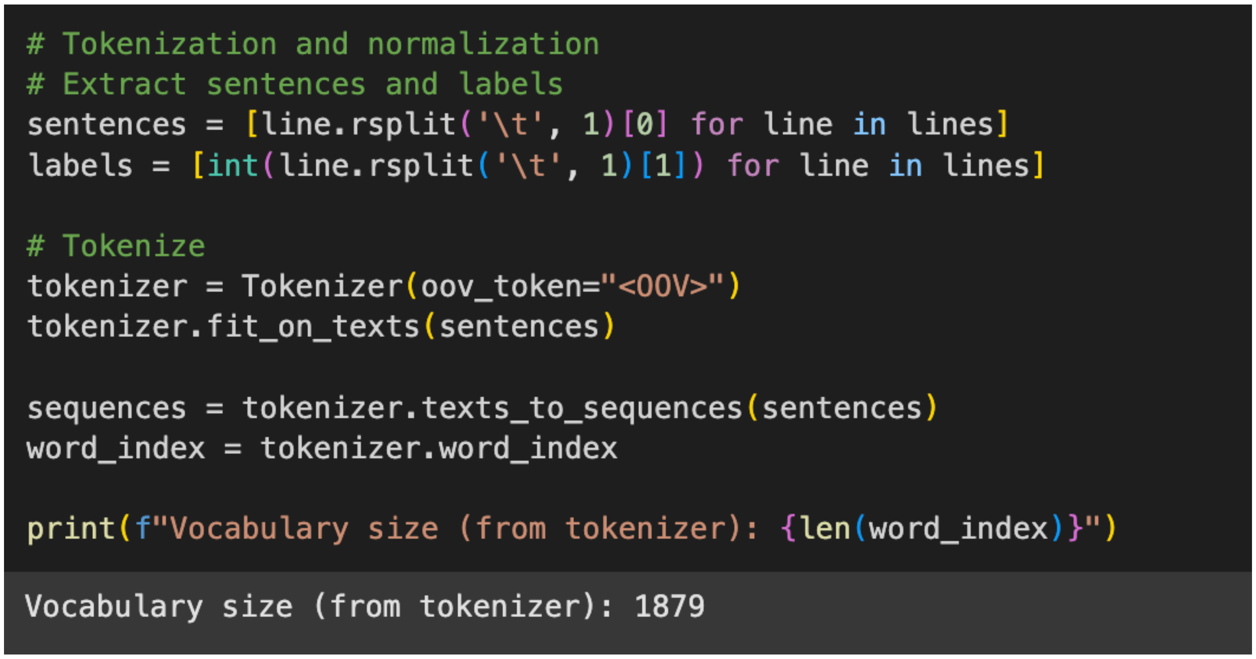
Lastly, having a statistical justification for maximum sequence length helps optimized both performance and training time. Neural networks require inputs of the same length, using the 90th percentile ensures most of your data is preserved without overdoing padding or truncation. In this case, the mean, max, and 90th percentile of the tokenized sentence lengths was computed. This helped identify a reasonable maximum length that captures most of the data without excessing padding or truncation. This corresponds to the 90th percentile, capturing most sentences without excessive padding.

Performing these EDA steps before modeling helps you clean your input text, optimize memory and model size, ensure consistency in preprocessing, and justify choices for architecture and preprocessing. Skipping this step in your cleaning process could lead to poor model accuracy, inefficient training, and misinterpretation of results.

B2, Tokenization Process Goals

The goals of the tokenization process is that tokenization transforms raw text into a sequence of tokens (words or indices), allowing it to be converted into numerical form for input into a neural network. It also standardizes vocabulary, handles unknown words, and enables embedding.

TensorFlow's Tokenizer with an out-of-vocabulary token was used to tokenize the text. The packages used consisted of nltk.tokenize and tensorflow.keras.preprocessing.text. During this process, the text was converted to lowercase before preprocessing, and the text was converted into integer sequences representing word indices.



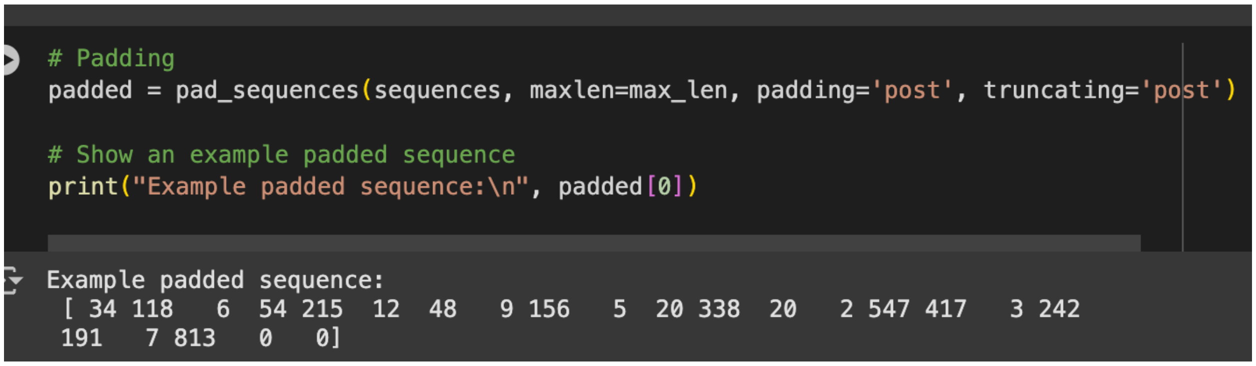
```
# Tokenization and normalization
# Extract sentences and labels
sentences = [line.rsplit('\t', 1)[0] for line in lines]
labels = [int(line.rsplit('\t', 1)[1]) for line in lines]

# Tokenize
tokenizer = Tokenizer(oov_token="<OOV>")
tokenizer.fit_on_texts(sentences)

sequences = tokenizer.texts_to_sequences(sentences)
word_index = tokenizer.word_index

print(f"Vocabulary size (from tokenizer): {len(word_index)}")
```

```
Vocabulary size (from tokenizer): 1879
```

B3, Padding Standardization Process

In natural language processing (NLP), padding is the process of adding extra values to a sequence so that all sequences are the same length. To feed two sentences into a neural network, both must be the same length, padding solves this by ensuring every input sequence has the same number of tokens.

For this analysis, to ensure consistent input shape, sequences were padded to the same length (max_len = 23), which helps feed fixed-size input into the network. A padding and truncation direction of post were used to perform the padding after the actual sequence and cut off words at the end if too long. Using 'post' is usually better for interpretability in NLP. The screenshot below shows the padding performed and a single example padded sequence output.



```
# Padding
padded = pad_sequences(sequences, maxlen=max_len, padding='post', truncating='post')

# Show an example padded sequence
print("Example padded sequence:\n", padded[0])
```

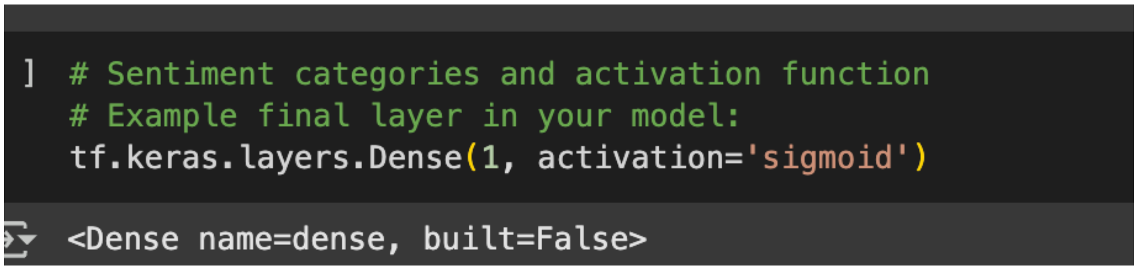
```
Example padded sequence:
 [ 34 118   6  54 215  12  48   9 156   5  20 338  20   2 547 417   3 242
 191   7 813   0   0]
```

B4, Sentiment Categories

In a sentiment analysis, the goal is to classify how someone feels based on text (i.e. a product review). The most common sentiment categories are a positive sentiment or a negative sentiment. This dataset includes positive (1) and negative (0) labels, or 2 sentiment categories, meaning the model only needs to decide if it is a positive or negative sentiment.

An activation function is what determines the final output of the model. For the output layer it shapes how the model makes its final decision. If it outputs a value between 0 and 1, this can be interpreted as the probability of the review being positive.

See code below determining 2 sentiment categories and the activation function using 'sigmoid'. This allows us to output a probability between 0 and 1, to determine a threshold.

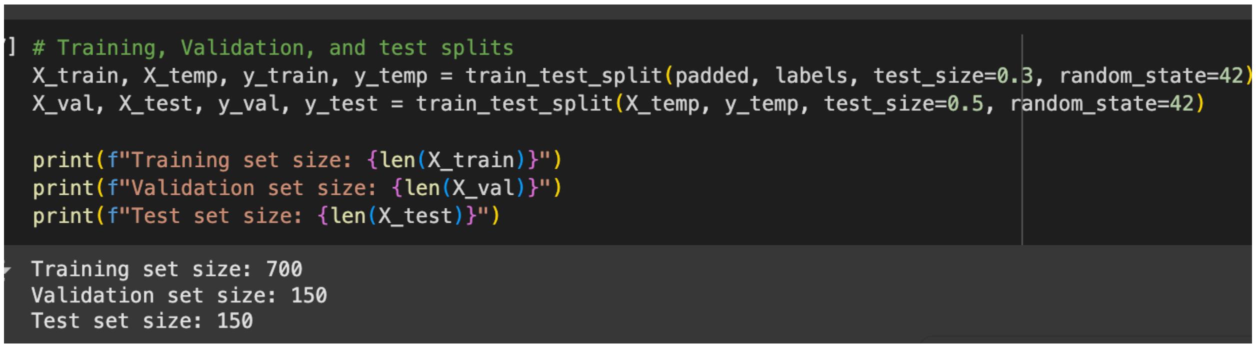


```
# Sentiment categories and activation function
# Example final layer in your model:
tf.keras.layers.Dense(1, activation='sigmoid')
```

```
<Dense name=dense, built=False>
```

B5, Data Preparation Steps

To prepare the dataset for analysis, the dataset was split into training, validation, and test sets using train_test_split() from sklearn. The industry standard split of 70/15/15 was followed. See code below outputting the train/test/split. The training set has 700 samples, validation set has 150 samples, and the test set has 150 samples. This ensures fair performance evaluation and generalizability of the model.



```
] # Training, Validation, and test splits
  X_train, X_temp, y_train, y_temp = train_test_split(padded, labels, test_size=0.3, random_state=42)
  X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

  print(f"Training set size: {len(X_train)}")
  print(f"Validation set size: {len(X_val)}")
  print(f"Test set size: {len(X_test)}")
```

```
Training set size: 700
Validation set size: 150
Test set size: 150
```

B6, Prepared Data

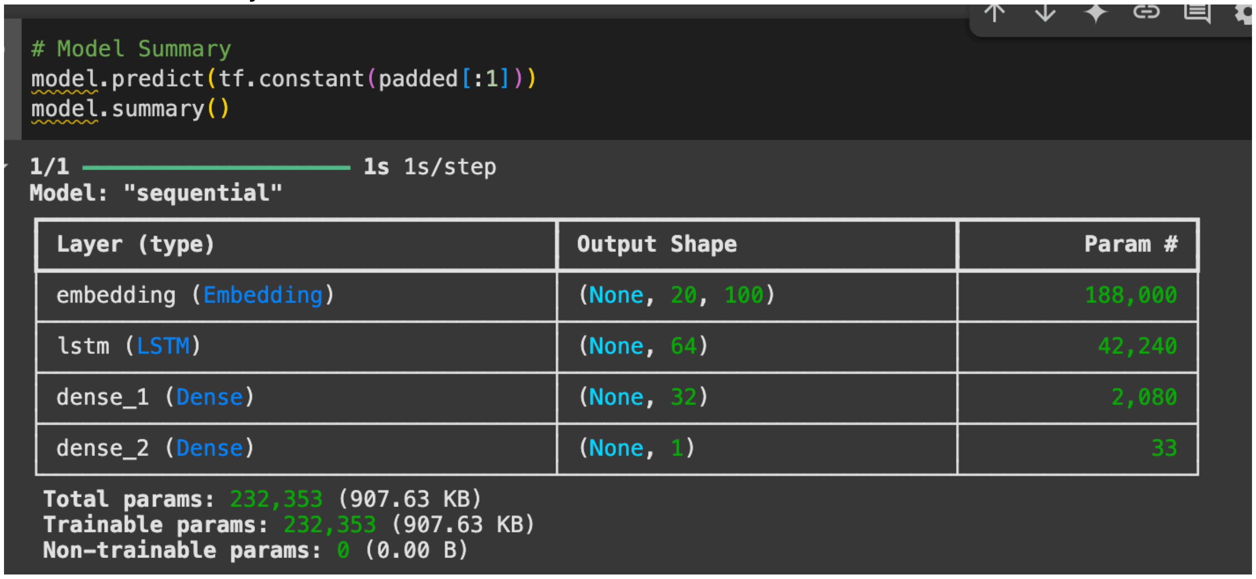
A prepared dataset outputted in CSV format can be found within the attached 'amazon_padded_train.csv' file.

C, Part 3: Network Architecture

C1, TensorFlow Model Summary Output

The neural network is built using TensorFlow's Sequential API and consists of an embedding layer, an LSTM layer, and two dense layers.

The model summary is as follows.



```
# Model Summary
model.predict(tf.constant(padded[:1]))
model.summary()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ embedding (Embedding)           │ (None, 20, 100)        │       188,000 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ lstm (LSTM)                     │ (None, 64)             │        42,240 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                 │ (None, 32)             │         2,080 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_2 (Dense)                 │ (None, 1)              │            33 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 232,353 (907.63 KB)
 Trainable params: 232,353 (907.63 KB)
 Non-trainable params: 0 (0.00 B)
```

C2, Data Layers and Parameters

The model consists of a total of four layers, the embedding layer, LSTM layer, dense layer (hidden) and dense layer (output).

The embedding layer converts each word index into a dense 100-dimensional vector, in this case there are 188,000 parameters. The LSTM layer processes the embedded sequences and learns temporal dependencies with 64 memory units, in this case there are 42, 240 parameters. The dense layer (hidden) is a fully connected layer with 32 nodes and ReLU activation provides a non-linear LSTM output transformation, in this case there are 2,080 parameters. Lastly, the dense layer (output) is a single-node output layer using a sigmoid activation to produce a binary probability (positive/negative), this has 33 parameters.

In total from all the layers, there are 232,353 parameters. All parameters are trainable, and the model has no non-trainable weights.

C3, Hyperparameters Choice

Hyperparameters are the configuration settings that are defined before training the model, these are not learned from the dataset instead they control how the model learns. The hyperparameters selected include activation functions, number of nodes per layer, loss function, optimizer, stopping criteria, and evaluation metric.

For activation functions, ReLU (Rectified Linear Unit) is used in the hidden dense layer for its ability to efficiently handle non-linearity and prevent vanishing gradients. Additionally, sigmoid is used in the output layer since it is ideal for binary classification, providing a probability between 0 and 1.

For the number of nodes per layer, 64 LSTM units were chosen to balance learning capacity and computational efficiency. And 32 nodes in the hidden dense layer are sufficient for the moderate complexity of this classification task.

Binary crossentropy is used as the loss function, which is standard and well-suited for binary classification tasks with sigmoid output.

Adam optimizer was chosen for its adaptive learning rate and strong performance across many NLP and text classification tasks.

To identify stopping criteria, early stopping can be implemented using the validation loss with a patience value (e.g., 3 epochs) to prevent overfitting.

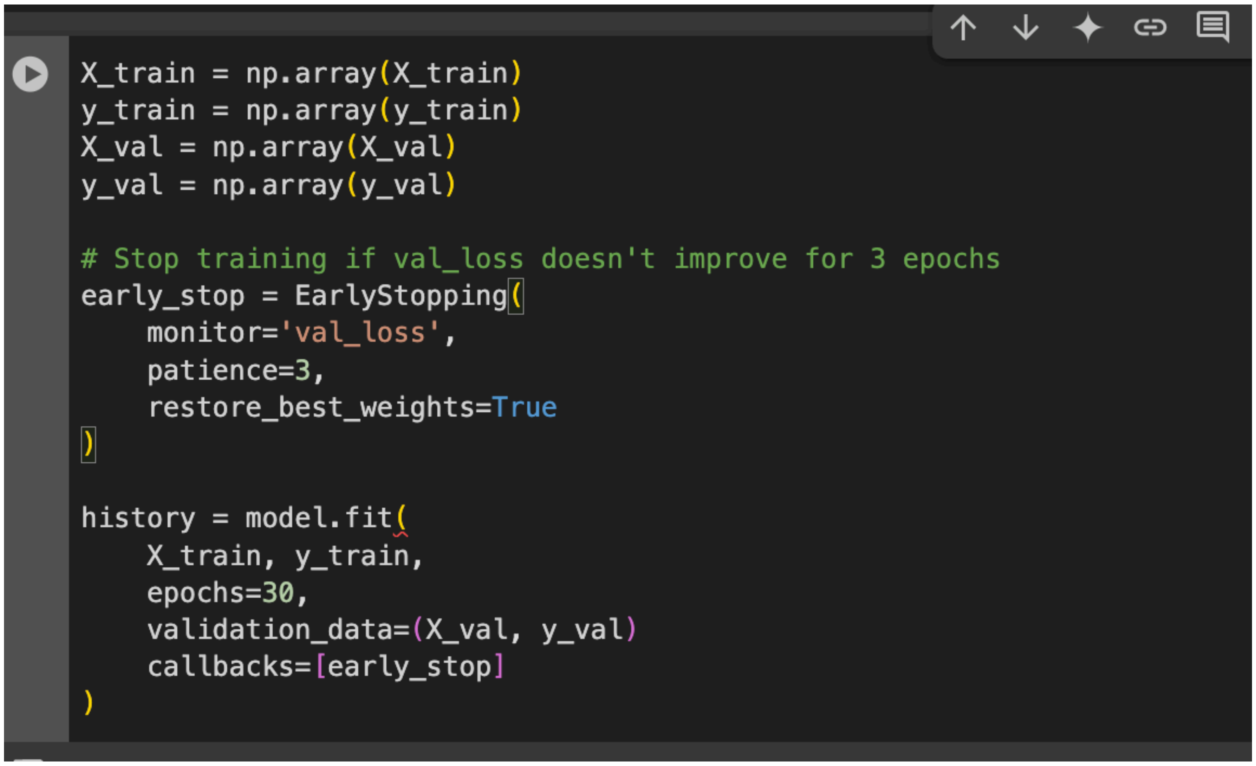
Lastly to determine an evaluation metric, accuracy is used because the dataset is balanced (equal positive and negative examples), making accuracy a fair and interpretable evaluation metric.

D, Part 4: Model Evaluation

D1, Impact of Training Data

Stopping criteria, like early stopping, helps prevent overfitting by halting training once the model stops improving on the validation set. Early stopping was implemented to monitor the validation loss and halt training when no improvement was observed for 3 consecutive epochs. This prevents unnecessary training and reduces overfitting.

The train data and values were passed through a NumPy array to properly be executed in early stopping. Training was configured for a maximum of 30 epochs but early stopping halted training after 8 epochs. See early stopping code executed below.



```python
X_train = np.array(X_train)
y_train = np.array(y_train)
X_val = np.array(X_val)
y_val = np.array(y_val)

# Stop training if val_loss doesn't improve for 3 epochs
early_stop = EarlyStopping(
    monitor='val_loss',
    patience=3,
    restore_best_weights=True
)

history = model.fit(
    X_train, y_train,
    epochs=30,
    validation_data=(X_val, y_val)
    callbacks=[early_stop]
)
```

Now we can start to determine the model's performance across epochs.

D2, Model Fitness, and Overfitting Actions

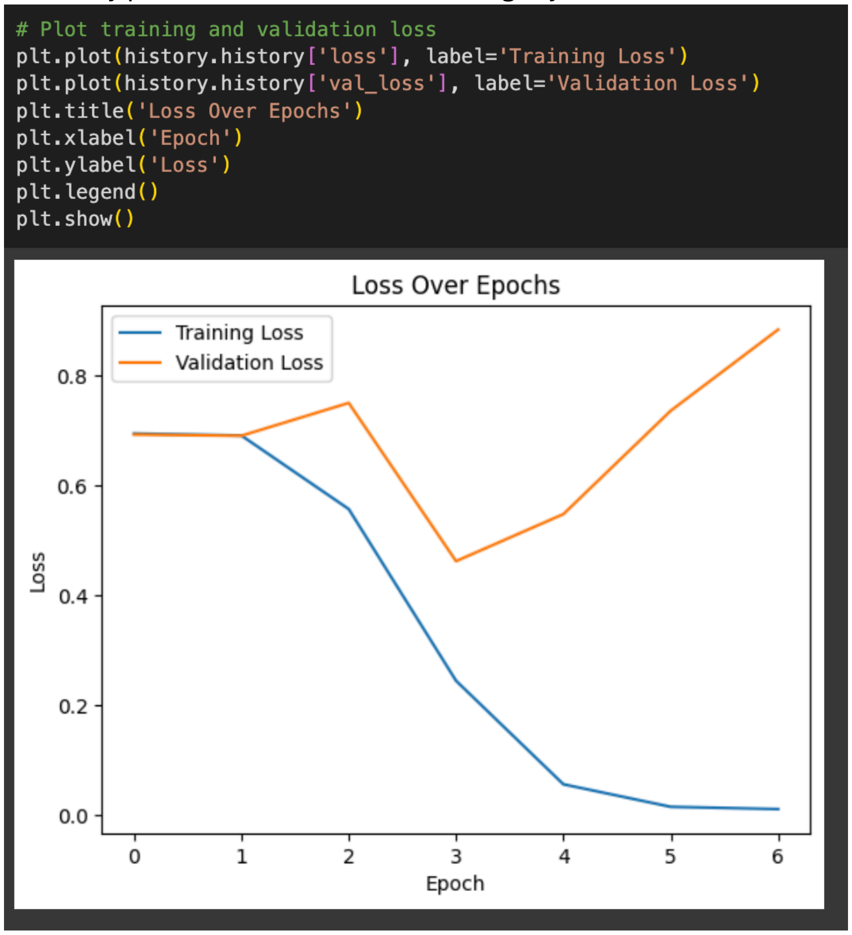
The model showed a clear improvement in both training and validation accuracy across the first few epochs. Training accuracy climbed from 51% to 99.7%, and validation accuracy improved from 51% to 82.7%.

Although training accuracy reached near perfection, the validation accuracy plateaued around 82-83%, suggesting some risk of overfitting, but not excessive. Early stopping likely helped minimize that risk by stopping before validation loss worsened significantly, as seen at epoch 8.

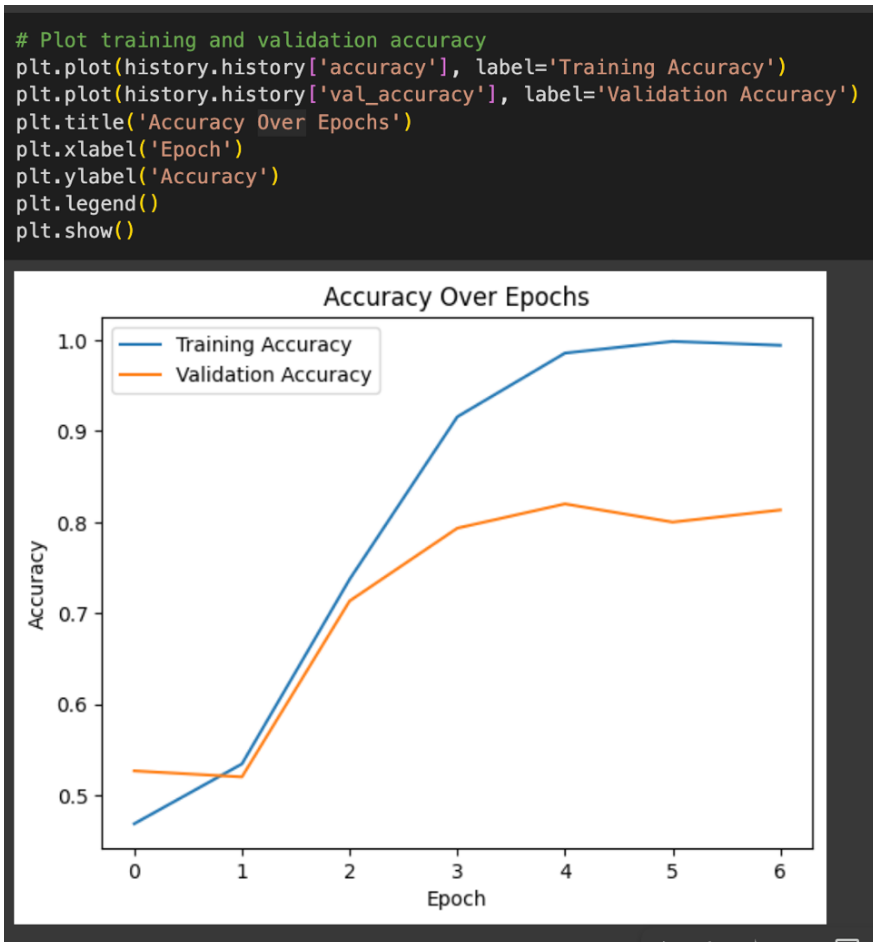
Overall, overfitting was addressed through early stopping, a moderate sized model, input padding and vocabulary normalization.

D3, Model Training Visualizations

Two line plots were generated to visualize both loss and accuracy over epochs. The loss chart below shows that training loss steadily decreased to near zero, and validation loss initially dropped but began increasing after epoch 4, indicating overfitting was beginning to occur. The epoch chart below shows that training accuracy steadily increased to 99.7%, and validation accuracy peaked around 82.6% then slightly declined. See charts below.



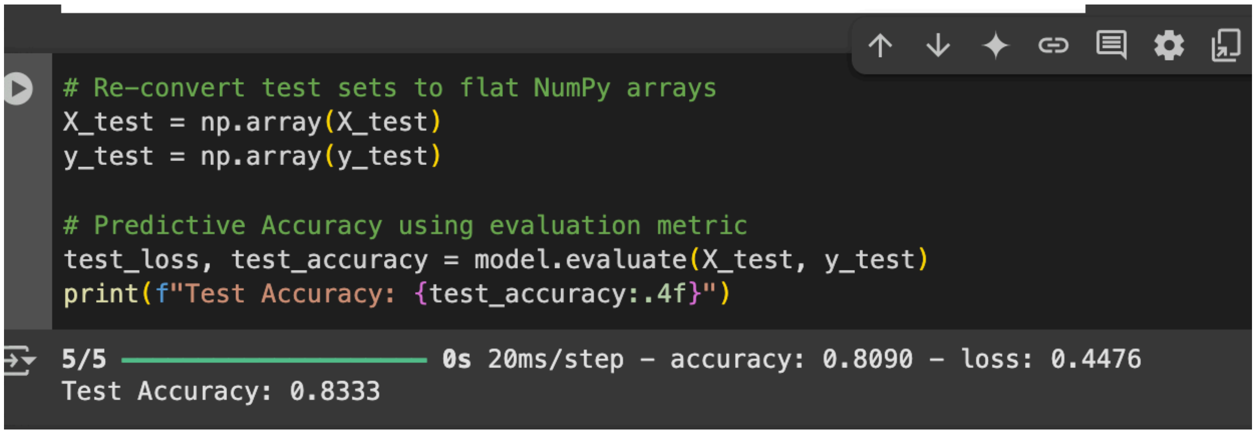
```
# Plot training and validation loss
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



```
# Plot training and validation accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

D4, Predictive Accuracy Discussion

The evaluation metric during training was accuracy, which is appropriate for this balanced binary classification task. To access generalization, the model was evaluated on the test set. The final test accuracy of the model was 83.33%, using accuracy as the evaluation metric. this is consistent with the validation accuracy observed during training (82.6%), which confirms that the model generalizes well to unseen data and did not overfit. Accuracy is an appropriate metric in this binary classification problem due to the balanced class distribution.



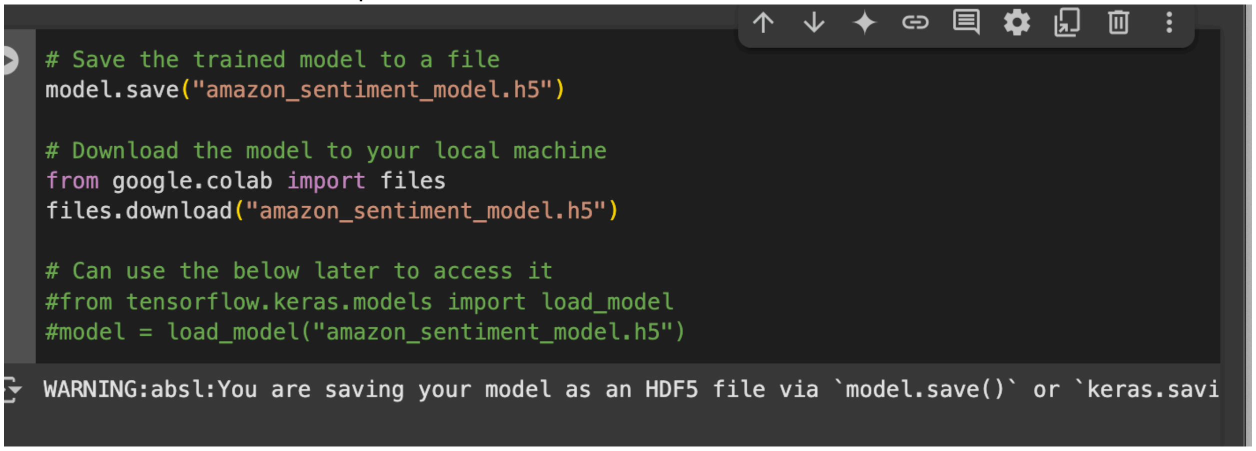
```
# Re-convert test sets to flat NumPy arrays
X_test = np.array(X_test)
y_test = np.array(y_test)

# Predictive Accuracy using evaluation metric
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_accuracy:.4f}")
```

```
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.8090 - loss: 0.4476
Test Accuracy: 0.8333
```

E, Part 5: Summary and Recommendations

E, Code

To save the trained model within Colab so it can be downloaded later or re-used you can use the below code. The output was saved to the file “amazon_sentiment_model.h5”.



```
# Save the trained model to a file
model.save("amazon_sentiment_model.h5")

# Download the model to your local machine
from google.colab import files
files.download("amazon_sentiment_model.h5")

# Can use the below later to access it
#from tensorflow.keras.models import load_model
#model = load_model("amazon_sentiment_model.h5")
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.savi
```

F, Functionality

The neural network is designed to classify amazon product reviews as positive or negative using sentiment analysis. The embedding layer converts each word into a numerical vector that captures semantic meaning. LSTM layer learns the sequence and context of words which is important for certain phrases. Dense layer adds non-linearity and abstraction. The output layer outputs a probability for binary classification.

Overall, the model functions from a combination of these layers being built. The embedding layer handles word-level semantics, the LSTM layer captures order and meaning across word sequences, and the architecture is compact and avoids overfitting while performing well. The models simplicity and efficiency make it well suited for real-time review classification tasks.

G, Course of Action

Based on the model’s strong performance (83.3% accuracy on test data), one course of action would be to deploy this neural network for automated sentiment tagging of Amazon reviews. This can help in filtering negative feedback automatically, routing complaints to customer service, and summarizing review sentiment for business intelligence.

Additionally, based on this model you should consider training on a larger dataset to increase generalization, experiment with dropout layers to reduce overfitting, and explore pre-trained embeddings for richer language understanding.

This model provides a reliable foundation for using AI to support decision-making in product analysis and customer feedback systems.

H, Part 6: Reporting

H, Jupyter Notebook Presentation

A prepared report has been completed using Jupyter Notebook. The outputted notebook presentation has been saved into an PDF document format which can be found within the attached ‘D213-Task2-Code.ipynb-Colab.pdf’ file.

I, Web Sources

I acknowledge that no segments of third-party sources were directly stated or copied from the web into this report.

J, Acknowledge the Sources

UCI Sentiment Labelled Sentences Data Set
Used to download the dataset and review its structure for preprocessing and training the sentiment classification model.
Webpage: https://archive.ics.uci.edu/dataset/331/sentiment+labelled+sentences

Kotzias, D. (2015). Sentiment Labelled Sentences [Dataset]. UCI Machine Learning Repository. https://doi.org/10.24432/C57604.