Eric Yarger

Sentiment Analysis Using Neural Networks

Research Question

Neural networks are computing systems that have interconnected nodes. These nodes work similarly to how neurons in the brain are connected and function. In industries and fields of all types, researchers and businesses are using neural net algorithms to find hidden patterns and correlations in their data. This process of finding hidden meaning has the potential to unlock great gains in performance and profitability. The question this analysis will answer is:

'Is it possible to create a model to analyze user sentiment from reviews using neural networks and NLP that will better enable our organization to identify customer satisfaction levels?'

Objectives and Goals

The overarching goal of this analysis is to create a model that can, with greater than 60% accuracy, predicts how a customer will score a review based on the words they use in the review. This goal will be met by addressing the following objectives:

- Make sure our review text is optimally preprocessed to be predictable and analyzable. This includes:
    - Ensuring that the model is provided with only the most relevant words from reviews by removing stopwords and special characters.
    - Breaking the stream of the sentence down into their individual meaningful elements by tokenizing.
    - Padding the reviews to be of equal length.
- Define the layers and parameters of the neural network model to accurately make predictions on the dataset.

Prescribed Network

A recurrent neural network (RNN) is an industry-relevant neural network that is trained to produce useful text classification predictions. RNNs are a class of artificial neural network (ANN) where connections create a cycle, allowing output from some nodes to exhibit an effect on following input into the same nodes (Recurrent neural network, n.d.).

Long short-term memory (LSTM), an RNN, is optimal for NLP sentiment analysis and text classification-related tasks. The name of LSTM refers to the fact that RNNs have both long-term and short-term memory. LSTM aims to provide a short-term memory for RNN, that can last for many timesteps.

LSTM has feedback connections, and can process entire sequences of data, such as speech, video, or text sequences (Long short-term memory, n.d.).

Data Exploration

Exploratory data analysis was performed on the dataset. Provided below is an explanation of each of the necessary elements.

1. Presence of unusual characters

    1.1. The presence of unusual characters can cause inefficiency in our sentiment analysis. Identification and removal of these characters is a necessary preprocessing step for this analysis. Provided below is a screenshot of the code used to identify characters from the dataset.

Presence of Unusual Characters

```
8]: # List characters in reviews
    # Code Reference (Elleh, 2022)

    characters = df['Review']
    list_chars = []
    for char in characters:
        for chars in char:
            if chars not in list_chars:
                list_chars.append(chars)
    print(list_chars)
```

```
['A', ' ', 'v', 'e', 'r', 'y', ',', 's', 'l', 'o', 'w', '-', 'm', 'i', 'n', 'g', 'a', 'b', 'u', 't', 'd', 'f', '.',
'N', 'h', 'c', 'k', 'p', '&', 'x', 'V', 'T', 'G', 'I', "'", 'W', 'S', 'L', 'J', 'B', 'F', 'M', 'H', 'C', '"', '\x96
', 'z', '?', 'q', 'Y', 'j', 'P', 'U', 'R', 'E', '1', '3', ';', '/', 'O', '2', '9', '0', ':', '*', 'D', 'Q', 'é',
'(', ')', '!', 'K', '$', '7', '5', 'Z', '\x85', '8', '+', '%', '4', 'å', '6', '\x97', 'X', 'ê', '#', '[', ']']
```

```
]:
```

    1.2. Keras' Tokenizer class is used to remove special characters by implementing the filter parameter. Additionally, all text in the reviews is converted to lowercase during this step. Provided below is a screenshot of the code used to perform this step.

Filter Special Characters

```
keras_token = Tokenizer(filters='\t\n})~!|#*=&<>;:+.{/,(?$^_-%@)[\\]', oov_token = '[UNK]', lower = True)
```

```
keras_token.fit_on_texts(characters)
```

2. Vocabulary size

2.1. Vocabulary size is the number of unique words in a dataset. The vocabulary size for this analysis is found using Keras' Tokenizer class. Once the Tokenizer is instantiated it is fit on df['Review']. The length of the word index is then printed out.

The vocabulary size for this dataset is 5171.

3. Proposed word embedding length

3.1. Word embedding is the position of the word in the learned vector space. Embedding length is the square root of the square root of the vocabulary size, rounded up to the nearest whole number. To find the word embedding length for this analysis the stated vocabulary size from above, 5171.

Therefore the word embedding length is 9.

4. Statistical justification for the chosen maximum sequence length

4.1. The chosen maximum sequence length for this analysis is 36. This was chosen by determining the maximum sequence length from the reviews. The minimum sequence length is 1, with the median length being 5. Padding will be used to address differentiations in sequence length. Padding makes each review contain the same length..

Tokenization

Tokenization is an important aspect of text preprocessing for our NLP application. Tokenization is the process of breaking down a stream of textual data into words, terms, sentences, symbols, or some other meaningful element, called tokens (Menzli, 2022). In the case of this analysis, the reviews will be broken down into their individual word components, also referred to as word tokenization. The generated tokens turn the unstructured review texts into a numerical data structure that is optimal for the creation of our neural network.

Before tokenization, the explored data set is split into training (64% of total data), test (20% of total data), and validation sets (16% of total data). Variables are assigned to the 'Review' column in the sets. The max number of features is assessed from section B1.3 above - where the vocabulary size was found to be 5171, which will be used as the 'num_words' argument in Tokenizer.

For the tokenization process in this analysis package Tokenizer from tensorflow.keras.preprocessing.text was implemented. Tokenizer can be broken down into the necessary

methods implemented for the tokenization processing in this analysis, explained in detail on TensorFlow's Tokenizer documentation page (Tf.keras.preprocessing.text.Tokenizer, n.d.).

- Instantiate class Tokenizer, assigned to variable 'tokenizer'. Filter out special characters and set out-of-vocabulary_token (argument oov_token) to replace out of value words with [UNK]. This filters out special characters, and makes sure that out of vocabulary words, maybe something like 'Rumplestilskin', are filtered out and replaced. Set argument lower = True to convert texts to lowercase.
- Take variable 'tokenizer' and implement method fit_on_texts on the list of text for the variable that holds the training reviews.
- Next, method texts_to_sequences transforms the text from the step above into a sequence of integers.
- To make sure the tokenization process worked as intended the tokens from the first 100 reviews in variable train_tokenized are printed.

  Provided below is a screenshot of the code used for the analysis.

Tokenization

```
# Set max number of features from counting number of unique review values from above
max_number_features = 5171

# Initialize Tokenizer
tokenizer = Tokenizer(num_words=max_number_features, filters='\t\n})~!|#*=&<>;:+.{/,(?$^_-%@)[\\]', oov_token = '[UNK]', lower = True)

# Fit tokenizer on text for training review
tokenizer.fit_on_texts(list(training_review))

train_tokenized = tokenizer.texts_to_sequences(training_review)
test_tokenized = tokenizer.texts_to_sequences(testing_review)
```

```
# Print small sample of Tokenized elements

print(train_tokenized[0:100])
```

```
[[1603, 1604, 2, 1605, 1606, 521, 126], [209, 522, 523, 28, 424, 970, 266, 425], [1607, 1608, 179, 180, 426, 267, 971, 2, 70, 972, 15, 427, 25, 15
7, 46, 303, 181, 1609], [50, 108, 268, 673], [17, 127, 2, 1610, 354, 146, 524, 973, 355, 674], [428, 2, 356, 182, 974, 675, 1611, 357, 269, 429],
[304, 85, 358], [158, 2, 305, 116, 4, 975, 4, 1612, 183], [210, 211], [86, 359, 976, 184, 1613, 1614], [676, 1615, 677], [2, 525, 117, 1616, 17, 2
3, 65, 678], [430, 17, 1617, 360], [212], [77, 679, 2, 680, 51, 181, 77, 431], [432, 25, 526, 681, 433, 434, 977, 26], [435, 159, 1618, 147, 436, 5
27, 85, 978, 306, 528, 3, 16, 23], [185, 361, 529, 2, 979, 1619, 270, 25, 1620, 307, 236, 980, 981], [1621, 109, 437, 530, 982, 682, 683, 983], [16
22, 3, 438, 1623, 984, 362, 271], [32, 3, 308, 1624, 684, 1625, 41, 531], [532, 1626, 439, 3, 78, 438, 1627, 1628, 47], [99, 77, 363, 440, 268, 2],
[100, 186, 1629, 1630, 985, 271, 986], [685, 3, 1631, 237, 686, 309, 1632, 987, 988, 272, 989, 128, 533], [160, 3, 238, 1633, 1634], [687, 129, 17,
161, 239], [990, 7, 17, 534, 239, 310], [441, 442, 27], [1635, 239], [535, 991, 535, 1636, 28, 535, 239], [89, 187, 239], [162, 187, 304, 992], [16
3, 687, 4, 12], [305, 687, 364], [48, 993, 240, 239, 1637], [239, 1638], [1639, 213, 188, 27], [688, 689], [365, 1640, 1641], [443, 444, 130, 1642,
71, 241, 690, 3, 42, 536, 1643, 994, 1644, 1645, 164, 2], [110, 148, 210, 995], [1646, 3, 22, 996, 997, 19, 998, 87, 214, 311, 2, 34, 1647, 41, 69
1, 41, 46, 312, 41, 692, 366, 1648], [90], [15, 693, 1649, 72, 1650], [1651, 20, 999, 31, 35, 66, 9, 1000, 1652, 1001, 694, 1002, 1653, 304], [43,
211, 1003], [55, 242, 88, 1004, 1654, 1655, 91, 56, 1005, 1656, 88, 36, 16, 57], [367, 695, 31, 33, 1657, 111, 189, 209], [3, 1658, 696, 445, 697,
1659, 434], [34, 368, 3, 33, 18, 1005, 1006, 1660, 3, 238, 22], [698, 7, 369, 430, 16, 313, 1007, 1661, 19, 1662, 7, 1663, 1664, 699, 16, 700, 19
0], [243, 1008, 1665, 27, 370], [701, 1666, 78, 1009, 1010, 101, 358, 191, 304, 537, 190], [159, 1011, 1012], [61, 371, 1013, 1667, 372, 7, 18], [5
2, 94, 43, 165, 373], [95, 374], [1014, 1668, 538, 539], [1669, 1670, 375, 1671, 1672, 1673, 1674], [376, 1675, 314, 190], [87, 7, 1676], [32, 446,
1677, 4, 215], [131, 702, 540, 703, 50], [70, 2], [7, 71, 95, 377, 1678, 378, 166, 1679, 1680], [95, 1681, 1682, 1683, 127], [102, 1015, 974, 91, 2
8], [2, 95, 1684, 1685], [373, 1686, 1687, 96, 315, 31, 1688, 447, 448, 1016, 165, 21, 447, 165, 4], [100, 541, 449, 4], [449, 149, 167, 13, 542,
7, 165, 101, 988, 1689], [1690, 1017, 1691, 543, 541, 132, 243, 542, 1692, 1693, 1018], [544, 6, 103], [379, 48, 448, 73, 358, 12, 118], [316, 169
4, 4, 1695, 165, 1696, 690], [216, 150, 1697, 1698, 704, 40, 29, 1699, 705, 1700, 133, 2, 23, 35, 706], [104, 2, 189, 707, 378, 1701, 6, 1006], [10
19, 1702, 6, 3], [37, 3, 1020, 7, 684, 1703, 1021], [34, 368, 545, 134, 33, 1704, 1705, 1706, 192, 708, 1022, 1023, 2, 1707, 546, 1708, 2, 450, 170
9], [44, 2, 36, 451, 112], [16, 2, 547, 1710, 1711], [168, 3, 119, 1712], [452, 1713, 1714, 1715, 160, 1024, 12], [87, 7, 65], [244], [709, 710, 2
1, 313, 1025, 711], [530, 317], [163, 1026, 45, 548, 129, 1716, 1717, 47, 18], [49, 73, 712], [1027, 62], [1718], [79, 374], [316, 166, 1719, 1720,
1721], [1722, 1028, 380, 3, 211, 1017, 12], [91, 449, 90, 549, 713, 28, 1723, 1029, 58], [94, 1724, 96, 1725, 453], [1726, 1727, 375, 1728, 713, 9
0, 12, 550, 240, 1729, 1030], [551, 2, 380, 94, 4, 1730, 1731, 1732, 1733, 30, 65, 1031, 1032, 193, 87, 7, 169]]
```

Padding Process

The padding process is an essential step in preparing the data for training. Padding is a form of masking where the masked steps are designated at the start or end of a sequence. This is necessary because the model only accepts sequences of the same length in a batch (Keras Team, 2020).

Where the individual reviews are not all the same length, package pad_sequences from tensorflow.keras.preprocessing.sequence is used to remedy this. pad_sequences takes the list of reviews as input, and returns a list of padded sequences. This means that each review in the list is standardized to be the same length.

The padded sequence length was determined by finding the longest length review in the list, which was 36, as identified during the data exploration stage of this analysis. The pad_sequence method has an argument that allows for padding to occur either before or after each sequence, designated as "pre" or "post". If no argument is specified it defaults to "pre". For this analysis, the default "pre" was used, which places the padding at the beginning of the sequence. Provided below is a screenshot of the code used for the padding process.

Padding

[32]:
```
# Pad Sequences
# Padding max_length chosen from finding max_length of review sequence from above


max_length = 36
X_train = pad_sequences(train_tokenized, maxlen=max_length)
X_test = pad_sequences(test_tokenized, maxlen=max_length)
```

The screenshot below shows an example of what a review sequence looks like from the analysis after padding. Note the padding is represented by the necessary number of zeros, '0', to pad the sequence to be the specified length of 36. To detail, this review was eight words long, with 28 'units' of padding placed before the sequence begins.

```
# Print Sample of prepared, padded, tokenized review

print(X_train[1])
```

```
[  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   0 208 521 522  27 423 969 265 424]
```

Categories of Sentiment

The scores for each review have been categorized into one of two categories: 0 or 1. Therefore, there are 2 categories used for sentiment analysis output in the neural network output layer.

'Sigmoid' is chosen for the fitting activation function for the final dense layer of the network. By default, if no activation function is specified linear activation is applied (Keras Team, n.d.). For the purpose of this analysis this wouldn't be the optimal activation function.

Both 'sigmoid' and 'softmax' were tested as selected attributes for final dense layer fitting activation. 'Softmax' is a multi-class, single label classification. For example, if the 'score' for a review ranged from 1-10, softmax would return an array of 10 probability scores that sum to 1. Each score would be the probability that the current review belongs to one of the 10 digit classes.

To contrast, selecting 'sigmoid' optimizes the fitting activation function for binary classification problems where the loss function is binary_crossentropy (Chengwei, 2018). The rating score in this analysis for each review is binary, either a 0 for negative review or a 1 for a positive review. Therefore, sigmoid was selected for the fitting activation function for the final dense layer of the network.

Steps to Prepare The Data

Provided below are the steps used to prepare the data.

1. Google Sheets, initial file preparation
    1.1. First step was to import the three text files into Google Sheets. Google Sheets 'Split Text To Columns' function was used to separate the Review and Score columns. Files yelp_labelled and amazon_cells_labelled separated with no errors. File imdb_labelled had ~10 reviews that required manual preparation. This was due to Sheets assigning the wrong cell type for half (for instance, Sheets interpreting a review of '10/10' as the date October 10), and half looked to be due to length restrictions of reviews. Both issues were identified on a case-by-case basis, and formatted to match the other data entries.

1.2. Once the three files were formatted correctly, they were cut and copied to a single file. Column names 'Review' and 'Score' were respectively added for column names. The resulting file was exported to .csv format. This file has two columns, 'Review' and 'Score' and contains 3000 cases. Column 'Review' has a datatype of 'object', Column 'Score' has a datatype of 'int64'. Column 'Score' contains 1500 '1' and 1500 '0' cases, with a mean of .5000000.

2. Set up environment

2.1. JupyterLab is used to access Jupyter Notebook, using Python.

3. Data file is read into the environment.

4. Data Exploration and cleaning.

4.1. Stopwords Removed.

4.1.1. Nltk's Stopwords package is used to remove stopwords. Stopwords are frequently used words, such as *the* or *it*, that do not add value by inclusion in the reviews for sentiment analysis.

4.2. WordCloud Generated.

4.2.1. Words from the reviews are visualized using WordCloud. The larger the text and more bold the word is in the visualization, the more frequently it is found in the dataset. This can be an important tool for visualizing the data for presentation to both technical and non-technical audiences.

4.3. Unusual Character Removal.

4.3.1. Unusual characters such as '$', '*', and ':', among many others, are removed from the reviews. This ensures that the dataset will contain only relevant characters to optimize the dataset for NLP.

4.4. Vocabulary size is established.

4.4.1. Tokenizer is fit_on_texts. Length of the word_index is printed out. This shows us that the vocabulary size for df['Review'], tokenized, is 5171.

4.5. Determine the max, min, median length of sequences.

4.5.1. Determining the max sequence size is an important step for optimizing our model parameters. If too large a size is chosen, when we go to pad the reviews there will be

unnecessary padding. This will make our model run slower. For large models, this would add a large amount of time and processing power necessary, with no tangible benefit to the output.

4.5.2. Max length of sequence in the reviews is 36, with a minimum of 1, and a median length of 5.

4.6. Train, Test, Validation Split.

4.6.1. Dataset is split into Train, Test, and Validation.

4.6.1.1. Training set is 64% of the total size of the dataset with 1920 cases.

4.6.1.2. Test set is 20% of the total size of the dataset with 599 cases.

4.6.1.3. Validation set is 16% of the total size of the dataset with 479 cases.

4.7. Variable assignment.

4.7.1. The test set column ['Review'] is assigned to variable 'testing_review'.

4.7.2. The training set column ['Review'] is assigned to variable 'training_review'.

4.7.3. The training set column ['Score'] is assigned to variable 'y'.

4.8. Tokenization.

4.8.1.1. The tokenization process is performed as detailed in section B2 of this analysis. Tokenizer is applied on training set variable with fit_on_texts() method.

4.9. Padding.

4.9.1.1. The padding process is performed as detailed in section B3 of this analysis.

4.10. Model Design.

4.10.1. Parameters Charting Function.

4.10.1.1. Function 'initialize_model' is defined to instantiate our model, the number of epochs, batch size, data appropriation location, and model checkpoint path.

4.10.1.2. Keras EarlyStopping is instantiated to execute early stopping criteria in the model.

4.10.1.3. Function 'chart' is defined to specify metrics, parameters, and arguments for plotting the model.fit data, stored in variable 'model_hist'.

4.11. Model Fit.

4.11.1. Training and testing data variables are NumPy arrays, and are prepared to be fit to the model.

Model Summary

Method model_name.summary() was used to provide a summary of the model. Provided below is a screenshot providing the complete output of the model summary.

```
# Set up Second Model
# Code Reference (PSS, 2021)

class LSTM2():
    def __new__(self):
        inp = Input(shape=(max_length, ))
        x = Embedding(5171, 128)(inp)
        x = LSTM(128, activation='sigmoid')(x)
        x = Dense(1, activation='sigmoid')(x)
        second_model = Model(inputs=inp, outputs=x)
        second_model.compile(loss='binary_crossentropy',optimizer='adam', metrics=['accuracy'])

        return second_model

lstm2 = LSTM2()
```

```
# Print Second Model Summary

print(lstm2.summary())
```

```
Model: "model_83"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_93 (InputLayer)       [(None, 36)]              0

 embedding_92 (Embedding)    (None, 36, 128)           661888

 lstm_86 (LSTM)              (None, 128)               131584

 dense_90 (Dense)            (None, 1)                 129

=================================================================
Total params: 793,601
Trainable params: 793,601
Non-trainable params: 0
_________________________________________________________________
```

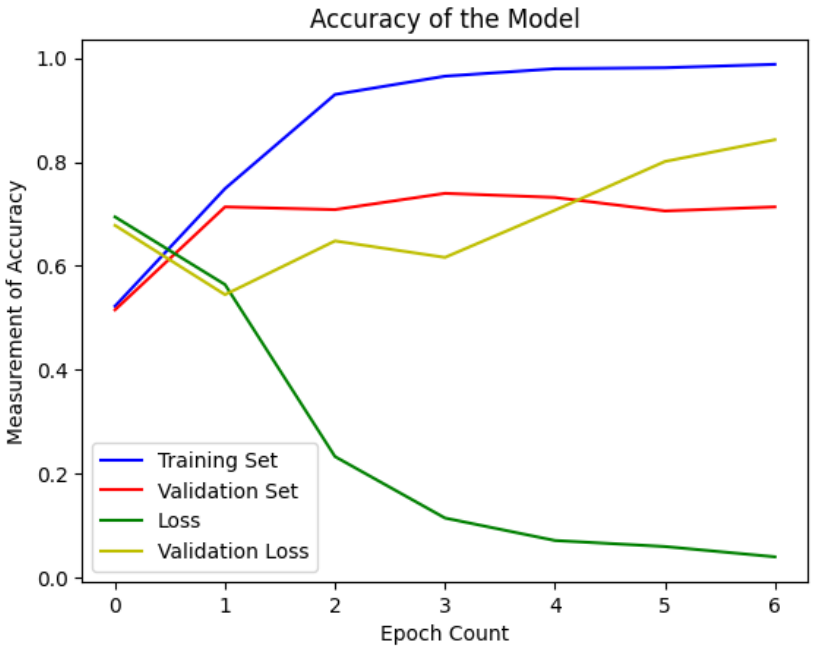
```
# Initialize and run Second Model

lstm2 = LSTM2()
history_lstm2 = initialize_model(lstm2, "lstm2", 10, 128, X_train, y, 0.2)
```

```
Epoch 1/10
128/128 [==============================] - 5s 32ms/step - loss: 0.6922 - accuracy: 0.5111 - val_loss: 0.6662 - val_accuracy: 0.6198
Epoch 2/10
128/128 [==============================] - 4s 29ms/step - loss: 0.4823 - accuracy: 0.8509 - val_loss: 0.7774 - val_accuracy: 0.6120
Epoch 3/10
128/128 [==============================] - 4s 31ms/step - loss: 0.1731 - accuracy: 0.9375 - val_loss: 0.6123 - val_accuracy: 0.7292
Epoch 4/10
128/128 [==============================] - 4s 29ms/step - loss: 0.0845 - accuracy: 0.9694 - val_loss: 0.6685 - val_accuracy: 0.7214
Epoch 5/10
128/128 [==============================] - 4s 29ms/step - loss: 0.0607 - accuracy: 0.9785 - val_loss: 0.7790 - val_accuracy: 0.7109
Epoch 6/10
128/128 [==============================] - 4s 29ms/step - loss: 0.0497 - accuracy: 0.9837 - val_loss: 0.8051 - val_accuracy: 0.7057
Epoch 6: early stopping
```

```
# Print Second Model Summary

chart(history_lstm2)
```



Network Architecture

Provided below is a summary of the network architecture.

- 1st Layer:
  - Layer type = Input
    - # of parameters = 0
- 2nd Layer:
  - Layer type = Embedding
    - # of parameters = 661888

- 3rd Layer:
    - Layer type = LSTM
        - # of parameters = 131584
- 4th Layer:
    - Layer type = Dense (Output)
        - # of parameters = 129

There are a total of four layers in the network. The total number of parameters in the network is 793,601.

Hyperparameters

Provided below is justification for the choice of hyperparameters for each listed element.

1. Activation functions
    a. Keras has eight available activations: softmax, softplus, softsign, relu, tanh, sigmoid, hard_sigmoid, and linear. When no activation function is specified for a layer, Keras applies 'linear' activation by default (Core layers - Keras documentation, n.d.).
    b. The activation function for the third and fourth layers is specified to be 'sigmoid'. The 'sigmoid' activation function, represented mathematically, is: sigmoid(x) = 1 / (1 + exp(-x)) (Tf.keras.activations.sigmoid, n.d.) . Sigmoid activation was chosen for these layers because the output is always binary: the output is either a 0 or a 1. Where the 'Score' for each review is either a 0 (negative review) or 1 (positive review) this activation function aligns the neural network with the data set.
2. Number of nodes per layer
    a. The number of nodes for the input and embedding layer were selected to match the maximum length of a single padded sequence, 36. For the third layer, LSTM, the number of nodes was selected from experimentation. I started with a low number of nodes and ran the model, and continued to add nodes until the resulting model accuracy from additional nodes began to diminish. The selected number of nodes for the third layer, LSTM, for this analysis is 128. The number of nodes in the final layer is determined by the chosen model configuration, which for this analysis the output layer has one node (D, 2010).

3. Loss function
    a. The purpose of loss function is to compute the quantity that a model seeks to minimize during training. There are three broad categories of loss functions available for selection in Keras: Probabilistic losses, Regression losses, and Hinge losses (Keras documentation: Losses, n.d.).
    b. For this analysis 'binary_crossentropy', a form of probabilistic losses class, was selected as the loss function. 'Binary_crossentropy' computes the cross-entropy loss between true labels and predicted labels. It is commonly used for binary classification applications, which aligns this loss-function with the analysis.
4. Optimizer
    a. An optimizer is a required argument for instantiating a Keras model. Optimizer 'Adam' was selected for this analysis. 'Adam' is a stochastic gradient descent method that is based on adaptive estimation of first and second order moments (Keras documentation: Adam, n.d.). This method was selected because it is efficient, has little memory requirement, and is well suited for a large range of problems.
5. Stopping criteria
    a. Keras' EarlyStopping is used for initiating stopping criteria in the model. 'Val_accuracy', which in this model is validation accuracy, is selected to be monitored. Patience is set to 3, meaning that the model will continue to run until three Epochs of decreasing 'val_accuracy' occur; at which point the model will be stopped early. Argument 'verbose' is set to 1, which prints the statement 'early stopping' when EarlyStopping has stopped the training early.
    b. Providing a stopping criteria is a smart method of controlling modeling time, computational load, while still ensuring that the model returns optimal results.
6. Evaluation metric
    a. In the model 'accuracy' is the metric selected to evaluate how well the neural network classifies reviews with the training dataset. The test dataset is used to determine if the model is at risk of overfitting. Provided below is a screenshot of the code used to execute this.

D4: Predictive Accuracy

```
# Using Keras Model Evaluation to
# test accuracy on Test data and test labels
# Code Reference (Keras - Model evaluation and model prediction, n.d.)

# initiate keras model.evaluate
# assign Test data and Test Labels
score = lstm2.evaluate(X_test, Y_test, verbose = 1)

# Print Test Accuracy
print('Test accuracy:', score[1])
```

```
19/19 [==============================] - 1s 13ms/step - loss: 1.1573 - accuracy: 0.6644
Test accuracy: 0.6644407510757446
```

The neural network's accuracy is .6644 on the withheld test dataset. This can be compared to the training accuracy of .9375 and val_accuracy of .7292. It can be seen that the model is overfitting on training data, and not performing as well on data it has never seen before.

Stopping Criteria

Stopping criteria is an important aspect in neural network training. A problem that arises with training neural networks is how to choose the correct number of training epochs to use. If too many epochs are used, the model can overfit the data. If too few epochs are the model can underfit the data. Both of these scenarios provide poor results (Vijay, 2019).

For this analysis, Keras EarlyStopping was implemented. Early stopping is a method that stops the training of the model once the model performance stops improving. Argument 'patience' for EarlyStopping sets the number of epochs to continue training after model performance stops improving before EarlyStopping stops training. In this analysis 'patience' was set to 3 for the model. Argument 'monitor' specifies which performance measure will be monitored to execute argument 'patience'. In this analysis argument 'monitor' tracked 'val_accuracy'. Argument 'verbose' specifies if the analyst wishes for EarlyStopping to display a message when EarlyStopping takes action. For this analysis 'verbose' was set to 1, indicating that messages were wanted. Provided below is a screenshot of the code used for EarlyStopping.

Stopping criteria: TF Keras EarlyStopping utilized.

monitoring 'val_accuracy', patience set to 3, verbosity is turned on

```
from tensorflow.keras.callbacks import EarlyStopping as EarlyStopping
```

```
# Keras EarlyStopping
# Code Reference (Keras documentation: EarlyStopping, n.d.)

callback = EarlyStopping(monitor='val_accuracy', patience=3, verbose=1)
```

Provided below is a screenshot showing the final training epoch, stopped on Epoch 6, with EarlyStopping's message.

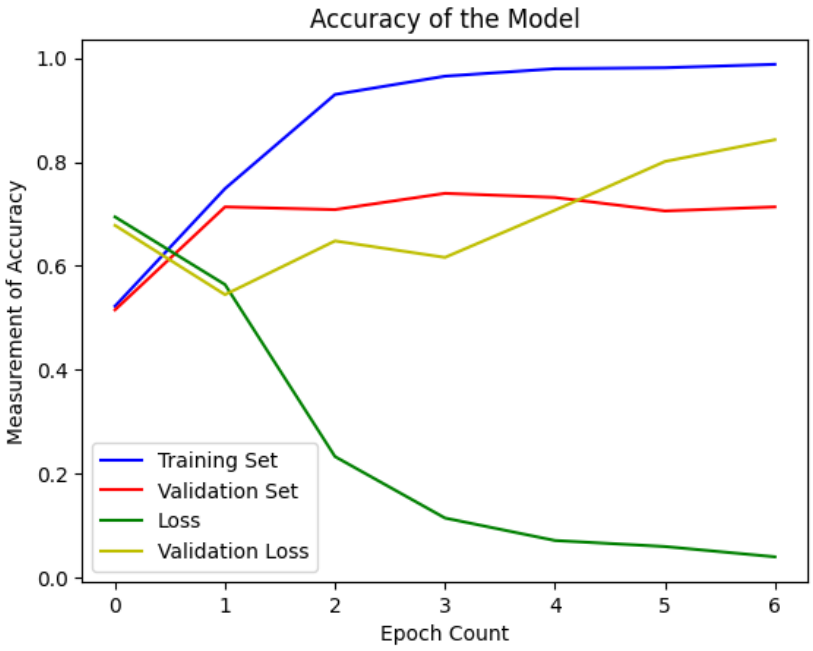
```
# Initialize and run Second Model

lstm2 = LSTM2()
history_lstm2 = initialize_model(lstm2, "lstm2", 10, 128, X_train, y, 0.2)
```

```
Epoch 1/10
128/128 [==============================] - 5s 32ms/step - loss: 0.6922 - accuracy: 0.5111 - val_loss: 0.6662 - val_accuracy: 0.6198
Epoch 2/10
128/128 [==============================] - 4s 29ms/step - loss: 0.4823 - accuracy: 0.8509 - val_loss: 0.7774 - val_accuracy: 0.6120
Epoch 3/10
128/128 [==============================] - 4s 31ms/step - loss: 0.1731 - accuracy: 0.9375 - val_loss: 0.6123 - val_accuracy: 0.7292
Epoch 4/10
128/128 [==============================] - 4s 29ms/step - loss: 0.0845 - accuracy: 0.9694 - val_loss: 0.6685 - val_accuracy: 0.7214
Epoch 5/10
128/128 [==============================] - 4s 29ms/step - loss: 0.0607 - accuracy: 0.9785 - val_loss: 0.7790 - val_accuracy: 0.7109
Epoch 6/10
128/128 [==============================] - 4s 29ms/step - loss: 0.0497 - accuracy: 0.9837 - val_loss: 0.8051 - val_accuracy: 0.7057
Epoch 6: early stopping
```

```
# Print Second Model Summary

chart(history_lstm2)
```



Training Process

For the model, the number of epochs is set to 10. Keras EarlyStopping is used, with patience set to 3. Setting the patience to 3 means that training will be stopped after 3 epochs with no improvement to val_accuracy, regardless of the number of epochs the model is set to train. EarlyStopping monitors 'val_accuracy' in the model, which represents validation accuracy.

In machine learning with Keras, 'Metrics' is used to evaluate the performance of the model. It's similar to loss function, but not used in the training process. There are a number of different metrics Keras provides to use, including: accuracy, binary_accuracy, categorical_accuracy, sparse_categorical_accuracy,

top_k_categorical_accuracy, sparse_top_k_categorical_accuracy, cosine_proximity, and clone_metric (Keras - Model compilation, n.d.).

The chosen model evaluation metric for the model in this analysis is 'accuracy'. The training early stops after 6 epochs, reaching peak val_accuracy at Epoch 3 of val_accuracy = .7292 and accuracy = .9375. In the line graph below titled 'Accuracy of the Model', the blue line 'Training Set' represents metric accuracy and the red line 'Validation Set' represents metric val_accuracy. Provided below are screenshots providing complete visualization of the model initialization, training, output, and charted visualization.

```
from tensorflow.keras.callbacks import EarlyStopping as EarlyStopping
```

```
# Keras EarlyStopping
# Code Reference (Keras documentation: EarlyStopping, n.d.)

callback = EarlyStopping(monitor='val_accuracy', patience=3, verbose=1)
```

```
# Function for initializing NLP LTSM model
# Sets data, epoch size, steps, and validation split
# Code Reference (PSS, 2021)

def initialize_model(model, model_name, number_epochs, batchsize, X_data, y_data, val_train):
    checkpoint_path = model_name+"_cp-{epoch:04d}.ckpt"
    checkpoint_dir = os.path.dirname(checkpoint_path)
    cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                     save_weights_only=True,
                                                     verbose=1)
    model_hist = model.fit(
        X_data,
        y_data,
        steps_per_epoch=batchsize,
        epochs=number_epochs,
        validation_split=val_train,
        callbacks=[callback]
    )
    return model_hist
```

```
# Function 'chart' assignment for plotting model results
# Code Reference (PSS, 2021)

def chart(model_hist):
    plt.plot(model_hist.history['accuracy'], 'b')
    plt.plot(model_hist.history['val_accuracy'], 'r')
    plt.plot(model_hist.history['loss'], 'g')
    plt.plot(model_hist.history['val_loss'], 'y')
    plt.title('Accuracy of the Model'),
    plt.xlabel('Epoch Count')
    plt.ylabel('Measurement of Accuracy')
    plt.legend(['Training Set', 'Validation Set', 'Loss', 'Validation Loss'], loc='lower left')
    plt.show()
```

```
# Set up Second Model
# Code Reference (PSS, 2021)

class LSTM2():
    def __new__(self):
        inp = Input(shape=(max_length, ))
        x = Embedding(5171, 128)(inp)
        x = LSTM(128, activation='sigmoid')(x)
        x = Dense(1, activation='sigmoid')(x)
        second_model = Model(inputs=inp, outputs=x)
        second_model.compile(loss='binary_crossentropy',optimizer='adam', metrics=['accuracy'])

        return second_model

lstm2 = LSTM2()
```

```
# Print Second Model Summary

print(lstm2.summary())
```

```
Model: "model_83"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_93 (InputLayer)       [(None, 36)]              0

 embedding_92 (Embedding)    (None, 36, 128)           661888

 lstm_86 (LSTM)              (None, 128)               131584

 dense_90 (Dense)            (None, 1)                 129

=================================================================
Total params: 793,601
Trainable params: 793,601
Non-trainable params: 0
_________________________________________________________________
```

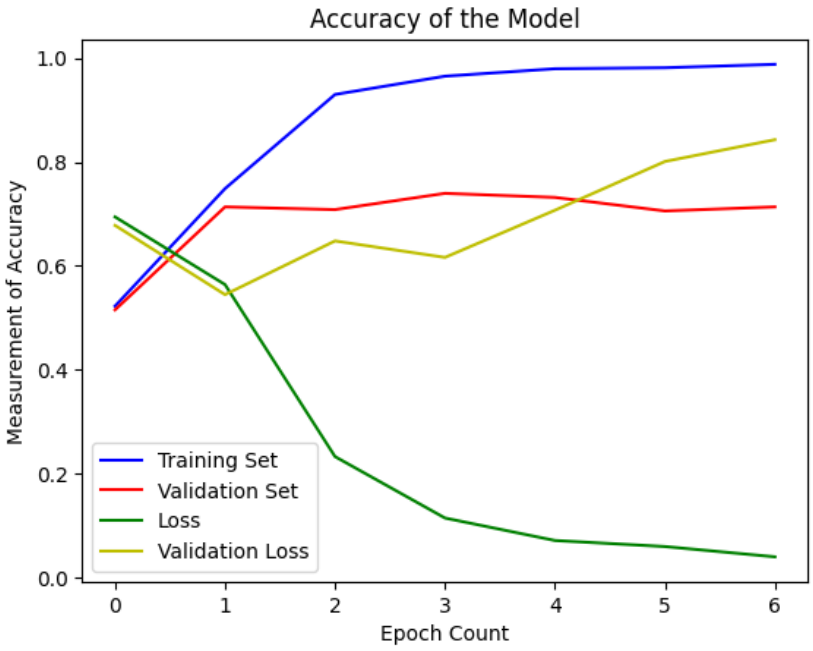
```
# Initialize and run Second Model

lstm2 = LSTM2()
history_lstm2 = initialize_model(lstm2, "lstm2", 10, 128, X_train, y, 0.2)
```

```
Epoch 1/10
128/128 [==============================] - 5s 32ms/step - loss: 0.6922 - accuracy: 0.5111 - val_loss: 0.6662 - val_accuracy: 0.6198
Epoch 2/10
128/128 [==============================] - 4s 29ms/step - loss: 0.4823 - accuracy: 0.8509 - val_loss: 0.7774 - val_accuracy: 0.6120
Epoch 3/10
128/128 [==============================] - 4s 31ms/step - loss: 0.1731 - accuracy: 0.9375 - val_loss: 0.6123 - val_accuracy: 0.7292
Epoch 4/10
128/128 [==============================] - 4s 29ms/step - loss: 0.0845 - accuracy: 0.9694 - val_loss: 0.6685 - val_accuracy: 0.7214
Epoch 5/10
128/128 [==============================] - 4s 29ms/step - loss: 0.0607 - accuracy: 0.9785 - val_loss: 0.7790 - val_accuracy: 0.7109
Epoch 6/10
128/128 [==============================] - 4s 29ms/step - loss: 0.0497 - accuracy: 0.9837 - val_loss: 0.8051 - val_accuracy: 0.7057
Epoch 6: early stopping
```

```
# Print Second Model Summary

chart(history_lstm2)
```



Fit

Overfitting is when a model learns the training dataset so accurately that it doesn't make accurate predictions on data that it hasn't seen. Fitness of the model can be assessed by comparing accuracy metrics on the data used for training, validation, and accuracy withheld to test on. From D2 above we see that at Epoch 3 the accuracy of the model was .9375 and the validation accuracy was .7292. This means the model was overfitting at this stage by .2083. Found in greater detail in section D4 of this analysis, accuracy on the test data is .6795. This means the model, when stopped, was overfitting the training data.

Measures that were taken to address overfitting included:

- Creating different models to compare and contrast how changes to parameters and layers affect accuracy, val_accuracy, and overfitting of the model
- Contrasting model accuracy with validation accuracy to measure when the model began to overfit the training data.
- Using Keras EarlyStopping, monitoring val_accuracy (representative of validation accuracy), with patience set to 3, to stop training once validation accuracy didn't increase for three Epochs. This ensured the model didn't continue to run, and overfit on accuracy alone, when actual predictive accuracy of the model was in decline.
- Hold-out data. 20% of the original data set was held out for evaluation and testing. Accuracy on the test data was .6795, which is much lower than the training data accuracy. This indicates the model was overfitting. This discrepancy in generalized accuracy is the reason Hold-out data is so important.

Additional measures that weren't implemented in this analysis but can address model overfitting include:

- Starting with a smaller network and increasing the capacity of the network as it improves by adding more layers, and more nodes to layers (Elleh, 2022).
- Providing more data for training and optimizing the network further.
- Further optimization model parameters.
- Keras API supports weight constraints. Weight regularization methods introduce a penalty to the loss function when the neural network is training. This encourages the network to use smaller weights. Smaller weights can result in a more stable model, which is reflected in better performance in model predictions and lower overfitting potential (Brownlee, 2019).
- Cross-validation. Unlike hold-out, cross validation allows for all of the data set to eventually be used for training (Lin, 2020).
- Remove layers. Sometimes a more complex model will overfit data, compared to a model with less layers.

Predictive Accuracy

Provided below is a screenshot showing Test accuracy from the coded portion of the analysis.

D4: Predictive Accuracy

```
# Using Keras Model Evaluation to
# test accuracy on Test data and test labels
# Code Reference (Keras - Model evaluation and model prediction, n.d.)

# initiate keras model.evaluate
# assign Test data and Test Labels
score = lstm2.evaluate(X_test, Y_test, verbose = 1)

# Print Test Accuracy
print('Test accuracy:', score[1])
```

```
19/19 [==============================] - 0s 15ms/step - loss: 0.8427 - accuracy: 0.6795
Test accuracy: 0.6794657707214355
```

These calculations utilize Keras model.evaluate method to check whether the model is best fit for the given problem and corresponding data (Keras - Model evaluation and model prediction, n.d.). The keras model.evaluate method takes the Test data and Test score values (test data labels) and evaluates model accuracy. This provides a statistically sound measurement of test accuracy.

The evaluation metric chosen from section D2 was 'accuracy'. In Keras, 'accuracy' calculates how often predictions equal the labels. Two local variables are created by the metric, 'total' and 'count' that are used to compute the frequency that labeled predictions match labeled true values (Keras documentation: Accuracy metrics, n.d.). Accuracy is important to evaluate on the analysis test data set .

The test set is data that was held aside, data the model hasn't made predictions on. Test accuracy on the test set was .6795. This means that the model accurately predicted the correct Score on the test data the model has never seen with 67.95% accuracy.

Code

The Keras API has methods that make it possible to save the architecture, layers, how layers are connected, the state of the model, an optimizer, and the specified losses and metrics to disk. To save the model, Keras model.save('path/to/location') is used (Save and load Keras models, n.d.). Provided below is a

screenshot from the accompanying Jupyter Notebook with all executed code, specifically section E: Save the Model:

Section E: Save the Model

Load the Model and make sure it is functional

```
]: lstm2.save("second_model")
```

```
INFO:tensorflow:Assets written to: second_model\assets
INFO:tensorflow:Assets written to: second_model\assets
```

```
]: from keras.models import load_model
reconstructed_model = load_model("second_model")
```

```
]: saved_model = initialize_model(reconstructed_model, "reconstructed_model", 10, 128, X_train, y, 0.2)
```

```
Epoch 1/10
128/128 [==============================] - 5s 32ms/step - loss: 0.0418 - accuracy: 0.9857 - val_loss: 1.0471 - val_accuracy: 0.7161
Epoch 2/10
128/128 [==============================] - 4s 31ms/step - loss: 0.0413 - accuracy: 0.9844 - val_loss: 1.1163 - val_accuracy: 0.6979
Epoch 3/10
128/128 [==============================] - 4s 30ms/step - loss: 0.0233 - accuracy: 0.9909 - val_loss: 1.2014 - val_accuracy: 0.6953
Epoch 4/10
128/128 [==============================] - 4s 32ms/step - loss: 0.0157 - accuracy: 0.9948 - val_loss: 1.2665 - val_accuracy: 0.7057
Epoch 4: early stopping
```

The loaded model from the save is complete and accurate. All code for this section, and all executed code can be found in the accompanying PDF of the Jupyter Notebook used for this analysis.

Functionality

The model training set contains 1920 customer reviews. The model uses Long-Short-Term-Memory (LSTM) network, a modified version of Recurrent Neural Networks (RNN), for NLP sentiment analysis of the reviews. Network accuracy and validation set accuracy are assessed during modeling. The accuracy of the model is validated on the test data, which is data that the model has never seen before. The results of these tests are described in detail throughout this document.

Sentiment analysis in this model is binary, and the model parameters selected reflect this. This helps optimize the model to accurately classify and predict sentiment for scores for customer reviews being positive (score of 1) or negative (score of 0). Provided below is a screenshot of model predictions on test data. Both an f-score and a confusion matrix are provided.

Model Predictions

Confusion Matrix and F-Score Calculation

```
# Define Funciton for model prediction
# confusion matrix and f-score calculation
# Code Reference (PSS, 2021)

def score_func(model):
  prediction_metric = model.predict(X_test)
  y_pred = (prediction_metric > 0.5)

  y_test = df_test['Score']

  conf_matrix = confusion_matrix(y_pred, y_test)
  f1score = conf_matrix[0][0] / (conf_matrix[0][0] + 0.5 * (conf_matrix[0][1] + conf_matrix[1][0]))
  print('Model F1 Score: %.4f' % f1score)
  print("Model Confusion Matrix Calculation :\n", conf_matrix)

  return f1score
```

```
# Confusion Matrix and F-Score, Second Model


lstm2_score = score_func(lstm2)
```

```
19/19 [==============================] - 0s 14ms/step
Model F1 Score: 0.6943
Model Confusion Matrix Calculation :
 [[218 100]
 [ 92 189]]
```

The F-1 score of .6943 is used as a measure of model accuracy, with a higher F-1 score indicating more accuracy in model classification. In this instance the F-1 is assessing the model's ability to accurately predict a positive or negative score on the test dataset. An F-1 score of .6943 isn't great, but it's in line with the test accuracy of .6795 identified from evaluating the model in previous steps.

The confusion provides a detailed visualization for analyzing the accuracy and performance of the neural network model. For reference, the test dataset used contains 599 reviews. The confusion matrix shows model prediction performance classified into four metrics:

- True Positive
    - 218
- False Positive
    - 100
- False Negative
    - 92
- True Negative
    - 189

True positives and True negatives are when the model accurately predicted the score of a review from the data. False positives are when the model inaccurately indicates that the score was positive. False negatives are when the model inaccurately predicts that a score was negative.

The impact of this model is that it positively addresses the question, objective, and goals for this analysis: that it can predict scores from review sentiment analysis with > 60% accuracy. However, while the model is useful for predictive analysis at ~68% accuracy, this shows just how much room there is for improvement in the model's predictive ability.

Recommendations

It is recommended that this model be used by our organization for sentiment analysis in predicting if a review will be positive or negative, with reservations. With ~68% accuracy on test data the model predicts review scores with greater than random predictive capacity, but leaves room for much improvement. This model can serve as the initial version of the tool, and can be used by our organization for making operational, marketing, and strategic decisions. However, it is recommended that if the organization's leadership team wished to prioritize the implementation of this technology, that more labor and capital be reserved for tuning hyperparameters to yield greater model accuracy, or testing different NLP models.

Reporting

The neural network is created in a Jupyter Notebook accessed through JupyterLab using Python 3.7.13 as the language and version. JupyterLab is an industry-relevant IDE, with Jupyter Notebook serving as the web application used for creating this computational analysis. A PDF of the executed Jupyter Notebook is included with the submission of this report.

Sources for Third Party Code

Elleh. (2022, May 15). D213 T2 May 15 22. Retrieved from https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=cedbd86a-2543-4d9d-9b0e-aec4011a606d

Keras documentation: EarlyStopping. (n.d.). Retrieved from https://keras.io/api/callbacks/early_stopping/

Keras - Model evaluation and model prediction. (n.d.). Retrieved from https://www.tutorialspoint.com/keras/keras_model_evaluation_and_prediction.htm

PSS, A. (2021, May 5). Sentiment analysis using LSTM. Retrieved from https://medium.com/mlearning-ai/sentiment-analysis-using-lstm-21767a130857

Sources

Brownlee, J. (2019, August 6). A Gentle Introduction to Weight Constraints in Deep Learning. Retrieved from https://machinelearningmastery.com/introduction-to-weight-constraints-to-reduce-generalization-error-in-deep-learning/

Chengwei. (2018, January). How to choose last-layer activation and loss function. Retrieved from https://www.dlology.com/blog/how-to-choose-last-layer-activation-and-loss-function/

Core layers - Keras documentation. (n.d.). faroit. https://faroit.com/keras-docs/1.2.0/layers/core/

D. (2010, August 2). How to choose the number of hidden layers and nodes in a feedforward neural network? Cross Validated. https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw

Elleh. (2022, May 15). D213 T2 May 15 22. Retrieved from https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=cedbd86a-2543-4d9d-9b0e-aec4011a606d

Keras - Model evaluation and model prediction. (n.d.). Retrieved from https://www.tutorialspoint.com/keras/keras_model_evaluation_and_prediction.htm

Keras documentation: Accuracy metrics. (n.d.). Retrieved from https://keras.io/api/metrics/accuracy_metrics/

Keras documentation: Adam. (n.d.). Keras: the Python deep learning API. https://keras.io/api/optimizers/adam/

Keras documentation: Losses. (n.d.). Keras: the Python deep learning API. https://keras.io/api/losses/

Keras Team. (n.d.). Keras documentation: Dense layer. Retrieved from https://keras.io/api/layers/core_layers/dense/

Keras Team. (2020, April 14). Keras documentation: Understanding masking & padding. Retrieved from https://keras.io/guides/understanding_masking_and_padding/

Lin, D. C. (2020, June 7). 8 simple techniques to prevent Overfitting. Retrieved from https://towardsdatascience.com/8-simple-techniques-to-prevent-overfitting-4d443da2ef7d

Long short-term memory. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Long_short-term_memory

Menzli, A. (2022, July 21). Tokenization in NLP: Types, challenges, examples, tools. neptune.ai. https://neptune.ai/blog/tokenization-in-nlp

Recurrent neural network. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Recurrent_neural_network

Save and load Keras models. (n.d.). Retrieved from https://www.tensorflow.org/guide/keras/save_and_serialize

Tf.keras.activations.sigmoid. (n.d.). TensorFlow. https://www.tensorflow.org/api_docs/python/tf/keras/activations/sigmoid

Tf.keras.preprocessing.text.Tokenizer. (n.d.). TensorFlow. https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/Tokenizer

Vijay, U. (2019, September 7). Early stopping to avoid overfitting in neural network- Keras. Retrieved from https://medium.com/zero-equals-false/early-stopping-to-avoid-overfitting-in-neural-network-keras-b68c96ed05d9