**Part I:  Research Question**

1. Describe the purpose of this data analysis by doing the following:
   1. Can we use a Recursive Neural Network (RNN) to determine whether a comment is positive or negative in sentiment?
   2. Our RNN should be able to classify reviews into two groups. The first group is negative reviews which have a score less than three out of five; the second group is positive reviews which have a score of three or greater out of five.
   3. We will be using a Recurrent Neural Network in order to capture the sentiment in the ordering of the words rather than each review being represented as a count matrix of words used, often referred to as a bag-of-words model.

**Part II:  Data Preparation**

1. Summarize the data cleaning process by doing the following:
2. Perform exploratory data analysis on the chosen dataset, and include an explanation of each of the following elements:
   1. We will use a custom function to check for non-ASCII characters, any non-English characters or emojis, and print the offending texts to the console. If any are found, we will remove the offending words in the tokenization step.

Input:

def non\_ascii(text):

char\_set = string.printable

return all((True if x in char\_set else False for x in text))

for text in df['reviewText']:

if not non\_ascii(text):

print(text)

There is no output from this code-block, implying that there are no non-ASCII characters in any of the reviews in our data set. No dropping will be required in the tokenization step relating to non-ASCII characters.

* 1. We can evaluate the vocabulary size by tokenizing the text with the num\_words hyperparameter set to none and counting the word index.

Input:

tokenizer = Tokenizer(oov\_token="<OOV>")

tokenizer.fit\_on\_texts(df['reviewText'])

word\_index = tokenizer.word\_index

vocab\_size = len(word\_index) + 1

vocab\_size

Output:

29304

We see from the output that the vocabulary size is 29,304 words, including the <OOV> tag which represents words skipped by the tokenizer and labeled as being “Out Of Vocabulary.”

* 1. We set the word embedding length to one-quarter the size of the vocabulary, rounded to the nearest integer (Google TensorFlow, 2017).

Input:

embedding\_dim = int(np.round(vocab\_size\*\*0.25))

This code produces no output but stores the embedding size for use in the embedding later.

* 1. We will calculate the mean, median, and third quartile of the review length distribution, rounded to the nearest integer, and graph them in relation to the distribution.

Input:

review\_length = df['reviewText'].apply(lambda x: len(x))

mean\_length = np.mean(df['reviewText'].apply(lambda x: len(x)))

median\_length = np.median(df['reviewText'].apply(lambda x: len(x)))

q3 = np.quantile(review\_length, 0.75)

plt.hist(review\_length, bins=100, range=(0, 1000))

plt.vlines(mean\_length, ymin=0, ymax=1000, colors='red', label='Mean: ' + str(np.round(mean\_length)))

plt.vlines(median\_length, ymin=0, ymax=1000, colors='green', label='Median: ' + str(np.round(median\_length)))

plt.vlines(q3, ymin=0, ymax=1000, colors='purple', label='Third-Quartile: ' + str(np.round(q3)))

plt.legend()

Output:

Chart, histogram

Description automatically generated

We can see that the median is much lower than the mean, implying we have a small number of very long reviews. The third quartile is much closer to the mean than the median is. Because of this, we will set the maximum sequence length to the third quartile value, rounded to the nearest integer.

1. Before tokenization, we will divide the data into training and test data, retaining 60% of the data for training, allocating 20% for validation, and using the remaining 20% for testing.

Input:

split1 = round(len(df)\*0.6)

split2 = round(len(df)\*0.8)

train\_reviews = df['reviewText'][:split1]

train\_label = df['sentiment'][:split1]

test\_reviews = df['reviewText'][split1:split2]

test\_label = df['sentiment'][split1:split2]

val\_reviews = df['reviewText'][split2:]

val\_label = df['sentiment'][split2:]

training\_sentences = []

training\_labels = []

testing\_sentences = []

testing\_labels = []

val\_sentences = []

val\_labels = []

for row in train\_reviews:

training\_sentences.append(str(row))

for row in train\_label:

training\_labels.append(row)

for row in test\_reviews:

testing\_sentences.append(str(row))

for row in test\_label:

testing\_labels.append(row)

for row in val\_reviews:

val\_sentences.append(str(row))

for row in val\_label:

val\_labels.append(row)

We can then begin tokenization. We will use the same tokenizer instantiated in the vocabulary size section to tokenize our data, using the texts\_to\_sequences method of the tokenizer to tokenize each review as a sequence.

Input:

sequences = tokenizer.texts\_to\_sequences(training\_sentences)

testing\_sequences = tokenizer.texts\_to\_sequences(testing\_sentences)

val\_sequences = tokenizer.texts\_to\_sequences(val\_sentences)

print(sequences[0])

Output:

[5, 220, 6, 200, 8, 1969, 693, 11, 14, 102, 29, 3, 28, 51, 39, 346, 3, 127, 43, 6, 39, 89, 25, 18, 113, 50, 6, 146, 283, 475, 3, 54, 79, 25, 22, 759, 4, 57, 1262, 76, 79, 5, 33, 190, 9, 15, 440, 1065, 77, 404, 58, 1992, 716, 123, 31, 16, 480, 40, 284, 707, 4, 284, 707, 25, 22, 878, 103, 19, 6, 157, 8, 283, 16, 33, 26, 5192, 187, 29, 32, 235, 53, 77, 567, 92, 8, 10, 5, 52, 124, 243, 6, 86, 299, 56, 5772, 41, 334, 53, 7431, 756, 244]

The output shows us that a tokenized review becomes a list of numbers which can be read by our model later in the analysis.

1. We pad and trim the tokenized sequences using the pad\_sequences method.

Input:

trunc\_type = 'post'

padding\_type = 'post'

padded = pad\_sequences(sequences, maxlen=max\_length, truncating=trunc\_type, padding=padding\_type)

testing\_padded = pad\_sequences(testing\_sequences, maxlen=max\_length, truncating=trunc\_type, padding=padding\_type)

val\_padded = pad\_sequences(val\_sequences, maxlen=max\_length, truncating=trunc\_type, padding=padding\_type)

* 1. The padding and trimming of sequences adds or takes from the end, respectively, because we set the truncating and padding hyperparameters to post.
  2. Input:

print(padded[0])

Output:

A picture containing table

Description automatically generated

1. There will be two categories of sentiment, positive and negative. Positive sentiment is represented by a value of one in the variable sentiment and represents a value of three or greater in the variable overall from the initial dataset. Negative sentiment is represented by a value of zero in the variable sentiment and represents a value less than three in the variable overall from the initial dataset. We create the sentiment variable with the following code:

df['sentiment'] = df.overall.apply(lambda x: 0 if x in [1,2] else 1)

1. The data preparation is explained in detail in the previous sections. A quick summary is included here. First, we load the dataset and add a column to represent the binary categorical variable sentiment. We then check for non-ASCII characters in the reviews; because we found none, there is no required cleaning in this aspect. Next, the vocabulary and word embedding sizes are calculated followed by deciding upon a review length. We then split the data into training, testing, and validation data using 60% of the data for training, 20% for validation, and 20% for testing. Finally, the split data is tokenized and padded or trimmed depending on the length of the review.
2. The training, test, and validation data are provided as the csv files train.csv, test,csv, and val.csv. The labels are included as the csv files train\_lab.csv, test\_lab,csv, and val\_lab.csv.

**Part III:  Network Architecture**

1. Describe the type of network used by doing the following:
2. Input:

model = tf.keras.Sequential([

tf.keras.layers.Embedding(vocab\_size, embedding\_dim, input\_length=max\_length),

tf.keras.layers.LSTM(100),

tf.keras.layers.Dense(1, activation='sigmoid')

])

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

model.summary()

Output:

Text

Description automatically generated

1. Our model has three layers: an embedding layer, a long short-term memory (LSTM) layer, and a dense layer. The embedding layer is our input layer, accepting the sequences and outputting each sequence as an array of shape 512 by 13. This contains all of the word embeddings for the sequence and feeds these into the LSTM layer. There are 380,952 trainable parameters for this layer, allowing for a large amount of fine tuning as the model trains. The LSTM layer provides the recursion that makes this a recursive neural network and allows the model to interpret the meaning of the sequence in a sequential manner rather than as a group of unordered words. This layer compresses the data down to 100 outputs. This layer has 45,600 trainable parameters, further refining our model’s ability to predict sentiment accurately. Finally, our dense layer operates as the output layer, accepting the 100 outputs from the LSTM layer and outputting only one value. There are 101 parameters to tune for this layer, allowing for minimal tuning in the final steps of the prediction. This layer has a single output that activates with a sigmoid function and makes the binary classification of sentiment. We have a total of 426,653 parameters that are all trainable.
2. Justify the choice of hyperparameters, including the following elements:
   1. Our LSTM layer uses a hyperbolic tangent activation function, and our output layer uses a sigmoid activation function. These activation functions were chosen in accordance with the recommendations from Brownlee, 2021. This article recommends that LSTM hidden layers use either hyperbolic tangent or sigmoid activation functions and binary classifier output layers to use sigmoid activation functions. The article also recommends hidden layers have different activation functions than the output layer. Combining these recommendations, we arrived at the decision to use a hyperbolic tangent for the LSTM layer and a sigmoid for the output layer. Input layers do not require activation functions as we want all information from the input to be fed forward.
   2. Through trial-and-error, we found that decreasing below 100 nodes in the LSTM layer reduced accuracy on the training and validation data, implying the model was underfitting the data during training. When the number of nodes is increased above 100, the opposite problem presents itself and the training accuracy increases at the cost of validation accuracy, implying the model begins to overfit the training data.
   3. We use a binary cross entropy loss function because our output is a binary categorization model.
   4. The optimizer choice had little effect on accuracy in our trial-and-error attempts with different optimizers. The ADAM optimizer did reach optimal accuracy much faster than the other optimizers tested so was chosen for our final model.
   5. We set our stopping criteria to stop once the model has gone five epochs with no improvement in the loss of the validation. This prevents the model from overfitting to the training data while allowing for some fluctuation as the model attempts to optimize the parameters of the network.
   6. We select accuracy as our optimization metric to ensure that the network prioritizes making correct predictions as it tunes the parameters rather than reducing loss.

**Part IV:  Model Evaluation**

1. Evaluate the model training process and its relevant outcomes by doing the following:
2. By using early stopping parameters, we can prevent the model from overfitting the training data. By setting the number of epochs with no early stopping parameters, the model will continue to improve based on the evaluation metric, accuracy in our model, with no consideration for the effect on the validation data. The final model output is shown below.



1. Input:

acc = history.history['accuracy']

val\_acc = history.history['val\_accuracy']

loss = history.history['loss']

val\_loss = history.history['val\_loss']

epochs=range(len(acc))

plt.plot(epochs, acc, 'r', label='Training Accuracy')

plt.plot(epochs, val\_acc, 'b', label='Validation Accuracy')

plt.title('Training and validation accuracy')

plt.legend()

plt.figure()

plt.plot(epochs, loss, 'r', label='Training Loss')

plt.plot(epochs, val\_loss, 'b', label='Validation Loss')

plt.title('Training and validation loss')

plt.legend()

plt.figure()

Output:

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated

We can see from the above graphs that the accuracy and loss generally increase for the training set but decrease steadily for the validation as the model overfits the training data. We can also see that the model begins with a fairly strong accuracy score and improvements or decreases over time are minimal.

1. We can see that the model begins to overfit after the first ten epochs with no increase in accuracy over those first ten epochs. By setting an early stopping warning, we prevent the model from proceeding through more than five epochs with an increase in the loss on the validation data.
2. We can evaluate the predictive power of our model on the training and the test data, the latter of which has never been seen by the model.

Input:

\_, train\_acc = model.evaluate(padded, training\_labels\_final, verbose=0)

\_, test\_acc = model.evaluate(testing\_padded, testing\_labels\_final, verbose=0)

print('Train: %.3f, Test: %.3f' % (train\_acc, test\_acc))

Output:



We see that the model can predict 93.5% of the test cases, only slightly behind the 94.7% accuracy on the training data.

**Part V:  Summary and Recommendations**

1. Provide the code used to save the trained network within the neural network.  
   model.save\_weights('./checkpoints/my\_checkpoint')
2. Our neural network works in three stages, each stage correlating to a layer of the network. In the first stage, the sequences are passed to the embedding layer which embeds the text in a numeric format so that it can be read by the following layers. The data is then passed to the LSTM layer where our second stage can begin. In this stage, the data is passed through the LSTM layer which will retain some of the information from the first pass as the later words filter through the layer, allowing the network to capture the sentiment in the ordering of the words. Finally, our data is passed to the output layer which interprets the sentiment into a final binary classification. The output of this final phase is a probability that sentiment is positive. This probability is input to our sigmoid activation function which will return either zero, for negative sentiment, or one, for positive sentiment.
3. Our model accurately predicts the sentiment of user reviews. This model could be supplied with customer feedback on social media sites, such as twitter, to quickly identify unhappy customers and address their concerns in a public forum, both ensuring quality experiences for our customers and providing low-cost marketing. This is not the only place the model could be useful; any point of contact where customers are providing feedback can be filtered into positive and negative sentiment for prioritization.

**Part VI: Reporting**

1. The Jupyter notebook and pdf version are included as Task\_2.ipynb and Task\_2.pdf respectively.
2. Most of the development was guided, influenced, and at some times borrowed from, the following source.

Sucky, R. N. (2021, July 8). A complete step by step tutorial on sentiment analysis in Keras and tensorflow. Medium. Retrieved November 17, 2021, from https://towardsdatascience.com/a-complete-step-by-step-tutorial-on-sentiment-analysis-in-keras-and-tensorflow-ea420cc8913f.

1. Acknowledge sources, using in-text citations and references, for content that is quoted, paraphrased, or summarized.

Brownlee, J. (2021, January 21). How to choose an activation function for deep learning. Machine Learning Mastery. Retrieved November 18, 2021, from https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/.

Google TensorFlow. (2017, November 20). Introducing tensorflow feature columns. Google Developers Blog. Retrieved November 18, 2021, from https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html.