btt004_week8_assignment-solution

June 10, 2024

1 Assignment 8: Implement a Neural Network for Sentiment Analysis

[1]:
```
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
import tensorflow.keras as keras
import time
```

In this assignment, you will implement a feedforward neural network that performs sentiment classification. You will complete the following tasks:

1. Build your DataFrame and define your ML problem:
    - Load the book review data set
    - Define the label - what are you predicting?
    - Identify the features
2. Create labeled examples from the data set
3. Split the data into training and test data sets
4. Transform the training and test text data using a TF-IDF vectorizer.
5. Construct a neural network
6. Train the neural network
7. Compare the neural network model's performance on the training and validation data.
8. Improve the model's generalization performance.
9. Evaluate the model's performance on the test data.
10. Experiment with ways to improve the model.

For this assignment, use the demo Transforming Text into Features for Sentiment Analysis that is contained in this unit as a reference.

Note: some of the code cells in this notebook may take a while to run

1.1 Part 1. Build Your DataFrame and Define Your ML Problem

Load a Data Set and Save it as a Pandas DataFrame We will work with the book review data set that contains book reviews taken from Amazon.com reviews.

Task: In the code cell below, use the same method you have been using to load the data using `pd.read_csv()` and save it to DataFrame `df`.

You will be working with the file named "bookReviews.csv" that is located in a folder named "data_NLP".

[2]:
```
# YOUR CODE HERE

#SOLUTION:

filename = os.path.join(os.getcwd(), "data_NLP", "bookReviews.csv")
df = pd.read_csv(filename, header=0)
```

Inspect the Data Task: In the code cell below, inspect the data in DataFrame `df` by printing the number of rows and columns, the column names, and the first ten rows. You may perform any other techniques you'd like to inspect the data.

[3]:
```
# solution
print(df.shape)
print(list(df.columns))
df.head(10)
```

```
(1973, 2)
['Review', 'Positive Review']
```

[3]:

	Review	Positive Review
0	This was perhaps the best of Johannes Steinhof...	True
1	This very fascinating book is a story written ...	True
2	The four tales in this collection are beautifu...	True
3	The book contained more profanity than I expec...	False
4	We have now entered a second time of deep conc...	True
5	I don't know why it won the National Book Awar...	False
6	The daughter of a prominent Boston doctor is d...	False
7	I was very disapointed in the book.Basicly the...	False
8	I think in retrospect I wasted my time on this...	False
9	I have a hard time understanding what it is th...	False

Define the Label This is a binary classification problem in which we will predict whether a book review is a positive or negative one. The label is the `Positive Review` column.

Identify Features We only have one feature. The feature is the `Review` column.

1.2 Part 2. Create Labeled Examples from the Data Set

Task: In the code cell below, create labeled examples from DataFrame `df`. Assign the label to the variable `y`. Assign the feature to the variable `X`.

[5]:
```
# YOUR CODE HERE

# solution

y = df['Positive Review']
X = df['Review']
```

1.3 Part 3. Create Training and Test Data Sets

Task: In the code cell below, create training and test sets out of the labeled examples. Create a test set that is 25 percent of the size of the data set. Save the results to variables `X_train, X_test, y_train, y_test`.

[6]:
```
# YOUR CODE HERE

# solution
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,␣
 ↪random_state=1234)
```

1.4 Part 4: Implement TF-IDF Vectorizer to Transform Text

In the code cell below, you will transform the features into numerical vectors using `TfidfVectorizer`.

Task: Follow the steps to complete the code in the cell below:

1. Create a `TfidfVectorizer` object and save it to the variable `tfidf_vectorizer`.
2. Call `tfidf_vectorizer.fit()` to fit the vectorizer to the training data `X_train`.
3. Call the `tfidf_vectorizer.transform()` method to use the fitted vectorizer to transform the training data `X_train`. Save the result to `X_train_tfidf`.
4. Call the `tfidf_vectorizer.transform()` method to use the fitted vectorizer to transform the test data `X_test`. Save the result to `X_test_tfidf`.

[7]:
```
# 1. Create a TfidfVectorizer object
# YOUR CODE HERE

#Solution:
tfidf_vectorizer = TfidfVectorizer()


# 2. Fit the vectorizer to X_train
# YOUR CODE HERE
```

```
# Solution
tfidf_vectorizer.fit(X_train)


# 3. Using the fitted vectorizer, transform the training data
# YOUR CODE HERE

#Solution:
X_train_tfidf = tfidf_vectorizer.transform(X_train)

# 4. Using the fitted vectorizer, transform the test data
# YOUR CODE HERE


#Solution:
X_test_tfidf = tfidf_vectorizer.transform(X_test)
```

When constructing our neural network, we will have to specify the `input_shape`, meaning the dimensionality of the input layer. This corresponds to the dimension of each of the training examples, which in our case is our vocabulary size. Run the code cell below to see the vocabulary size.

[8]:
```
vocabulary_size = len(tfidf_vectorizer.vocabulary_)

print(vocabulary_size)
```

```
18558
```

1.5 Part 5: Construct a Neural Network

1.5.1 Step 1. Define Model Structure

Next we will create our neural network structure. We will create an input layer, three hidden layers and an output layer:

- Input layer: The input layer will have the input shape corresponding to the vocabulary size.
- Hidden layers: We will create three hidden layers, with 64, 32, and 16 units (number of nodes) respectively. Each layer will utilize the ReLU activation function.
- Output layer: The output layer will have 1 unit. The output layer will utilize the sigmoid activation function. Since we are working with binary classification, we will be using the sigmoid activation function to map the output to a probability between 0.0 and 1.0. We can later set a threshold and assume that the prediction is class 1 if the probability is larger than or equal to our threshold, or class 0 if it is lower than our threshold.

Use the same approach you have taken in this course to construct a feedforward neural network model using Keras. Do the following:

- Use the Keras `Sequential` class to group a stack of layers. This will be our neural network model object. Name your neural network model object `nn_model`.

- Use the `InputLayer` class to create the input layer.
- Use the `Dense` class to create each hidden layer and the output layer.
- After creating each layer, add it to the neural network model object `nn_model`.

[9]:
```
# 1. Create model object

# YOUR CODE HERE

# solution
nn_model = keras.Sequential()


# 2. Create the input layer and add it to the model object:

# YOUR CODE HERE

# SOLUTION:
input_layer = keras.layers.InputLayer(input_shape=(vocabulary_size,))
nn_model.add(input_layer)

# 3. Create the first hidden layer and add it to the model object:

# YOUR CODE HERE

# SOLUTION:
hidden_layer_1 = keras.layers.Dense(units=64, activation='relu')
nn_model.add(hidden_layer_1)

# 4. Create the second layer and add it to the model object:

# YOUR CODE HERE

# SOLUTION
hidden_layer_2 = keras.layers.Dense(units=32, activation='relu')
nn_model.add(hidden_layer_2)


# 5. Create the third layer and add it to the model object:

# YOUR CODE HERE

# SOLUTION:
hidden_layer_3 = keras.layers.Dense(units=16, activation='relu')
nn_model.add(hidden_layer_3)


# 6. Create the output layer and add it to the model object:
```

```
# YOUR CODE HERE

# SOLUTION
output_layer = keras.layers.Dense(units=1, activation='sigmoid')
nn_model.add(output_layer)


# Print summary of neural network model structure
nn_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 64)                1187776
_________________________________________________________________
dense_1 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_2 (Dense)              (None, 16)                528
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 17
=================================================================
Total params: 1,190,401
Trainable params: 1,190,401
Non-trainable params: 0
_________________________________________________________________
```

1.5.2 Step 2. Define the Optimization Function

Task: In the code cell below, create a stochastic gradient descent optimizer using `keras.optimizers.SGD()`. Specify a learning rate of 0.1. Assign the result to the variable`sgd_optimizer`.

[10]:
```
# YOUR CODE HERE


# SOLUTION

sgd_optimizer = keras.optimizers.SGD(learning_rate=0.1)
```

1.5.3 Step 3. Define the Loss Function

Task: In the code cell below, create a binary cross entropy loss function using `keras.losses.BinaryCrossentropy()`. Since our output will be a normalized probability between 0 and 1, specify that `from_logits` is `False`. Assign the result to the variable `loss_fn`.

[11]:
```
# YOUR CODE HERE


# SOLUTION

loss_fn = keras.losses.BinaryCrossentropy(from_logits=False)
```

1.5.4 Step 4. Compile the Model

Task: In the code cell below, package the network architecture with the optimizer and the loss function using the `nn_model.compile()` method. Specify the optimizer, loss function and the accuracy evaluation metric as arguments.

[12]:
```
# YOUR CODE HERE


# SOLUTION
nn_model.compile(optimizer=sgd_optimizer, loss=loss_fn, metrics=['accuracy'])
```

1.6 Part 6. Fit the Model on the Training Data

We will define our own callback class to output information from our model while it is training. Make sure you execute the code cell below so that it can be used in subsequent cells.

[13]:
```
class ProgBarLoggerNEpochs(keras.callbacks.Callback):

    def __init__(self, num_epochs: int, every_n: int = 50):
        self.num_epochs = num_epochs
        self.every_n = every_n

    def on_epoch_end(self, epoch, logs=None):
        if (epoch + 1) % self.every_n == 0:
            s = 'Epoch [{}/ {}]'.format(epoch + 1, self.num_epochs)
            logs_s = ['{}: {:.4f}'.format(k.capitalize(), v)
                      for k, v in logs.items()]
            s_list = [s] + logs_s
            print(', '.join(s_list))
```

Task: In the code cell below, fit the neural network model to the vectorized training data. Call the `fit()` method on the model object `nn_model` and specify the following arguments:

1. The training data `X_train_tfidf` and `y_train` as arguments. Note that `X_train_tfidf` is currently of type sparce matrix. The Keras `fit()` method requires that input data be of specific types. One type that is allowed is a NumPy array. Convert `X_train_tfidf` to a NumPy array using the `toarray()` method.
2. Use the `epochs` parameter and assign it the number of epochs.
3. Use the `verbose` parameter and assign it the value of 0.
4. We will use a portion of our training data to serve as validation data. Use the `validation_split` parameter and assign it the value `0.2`

5. Use the `callbacks` parameter and assign it a list containing our logger function: `callbacks=[ProgBarLoggerNEpochs(num_epochs, every_n=5)]`

Save the results to the variable `history`.
Note: This may take a while to run.

[14]:
```
t0 = time.time() # start time

num_epochs = 55 #epochs

# history = # YOUR CODE HERE


# SOLUTION
history = nn_model.fit(
    X_train_tfidf.toarray(),
    y_train,
    epochs=num_epochs,
    verbose=0, # disable the default progress bar
    validation_split=0.2,
    callbacks=[ProgBarLoggerNEpochs(num_epochs, every_n=5)],
)


t1 = time.time() # stop time

print('Elapsed time: %.2fs' % (t1-t0))
```

```
Epoch [5/ 55], Loss: 0.6899, Accuracy: 0.5359, Val_loss: 0.6910, Val_accuracy:
0.6149
Epoch [10/ 55], Loss: 0.6675, Accuracy: 0.6298, Val_loss: 0.6723, Val_accuracy:
0.6115
Epoch [15/ 55], Loss: 0.5418, Accuracy: 0.7194, Val_loss: 0.5666, Val_accuracy:
0.7027
Epoch [20/ 55], Loss: 0.4131, Accuracy: 0.8115, Val_loss: 0.4721, Val_accuracy:
0.7872
Epoch [25/ 55], Loss: 0.3260, Accuracy: 0.8631, Val_loss: 1.5219, Val_accuracy:
0.5000
Epoch [30/ 55], Loss: 0.0813, Accuracy: 0.9848, Val_loss: 0.5617, Val_accuracy:
0.7568
Epoch [35/ 55], Loss: 0.0100, Accuracy: 1.0000, Val_loss: 0.4396, Val_accuracy:
0.8074
Epoch [40/ 55], Loss: 0.0042, Accuracy: 1.0000, Val_loss: 0.4707, Val_accuracy:
0.8142
Epoch [45/ 55], Loss: 0.0025, Accuracy: 1.0000, Val_loss: 0.4929, Val_accuracy:
0.8277
Epoch [50/ 55], Loss: 0.0017, Accuracy: 1.0000, Val_loss: 0.5098, Val_accuracy:
0.8311
```

```
Epoch [55/ 55], Loss: 0.0013, Accuracy: 1.0000, Val_loss: 0.5212, Val_accuracy:
0.8108
Elapsed time: 17.22s
```

[15]:
```
history.history.keys()
```

[15]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

1.6.1 Visualize the Model's Performance Over Time

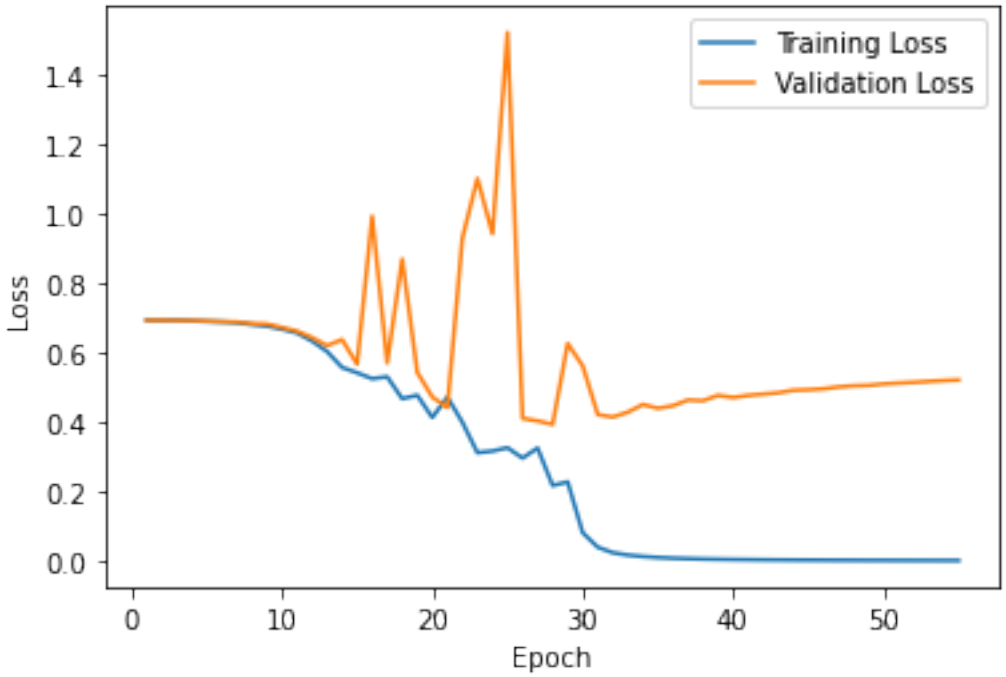
The code below outputs both the training loss and accuracy and the validation loss and accuracy. Let us visualize the model's performance over time:

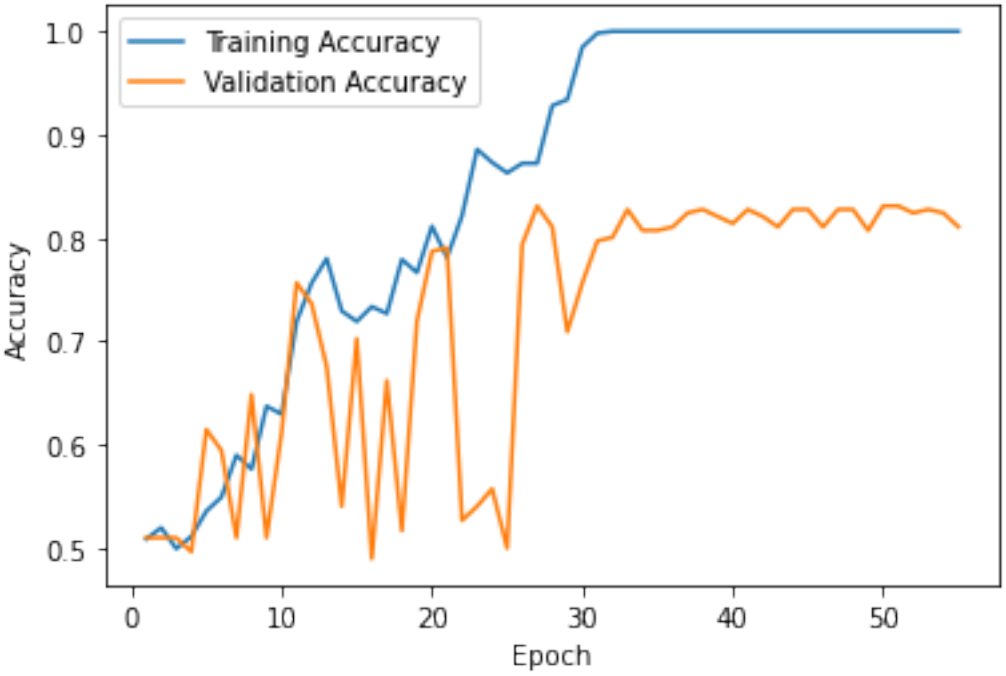
[16]:
```
# Plot training and validation loss
plt.plot(range(1, num_epochs + 1), history.history['loss'], label='Training␣
 ↪Loss')
plt.plot(range(1, num_epochs + 1), history.history['val_loss'],␣
 ↪label='Validation Loss')

plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()


# Plot training and validation accuracy
plt.plot(range(1, num_epochs + 1), history.history['accuracy'], label='Training␣
 ↪Accuracy')
plt.plot(range(1, num_epochs + 1), history.history['val_accuracy'],␣
 ↪label='Validation Accuracy')

plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```


Training Loss
Validation Loss
Loss
Epoch
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
10
20
30
40
50


Training Accuracy
Validation Accuracy
Accuracy
Epoch
0.5
0.6
0.7
0.8
0.9
1.0
0
10
20
30
40
50

1.7 Part 7. Evaluate the Model's Performance

1.7.1 Improve Model's Performance and Prevent Overfitting

Neural networks can be prone to overfitting. Notice that the training accuracy is 100% but the validation accuracy is around 80%. This indicates that our model is overfitting; it will not perform as well on new, previously unseen data as it did during training. We want to have an accurate idea of how well our model will generalize to new data. Our goal is to have our training and validation accuracy scores be as close as possible.

While there are different techniques that can be used to prevent overfitting, for the purpose of this exercise we will focus on two methods:

1. Changing the number of epochs. Too many epochs can lead to overfitting of the training dataset, whereas too few epochs may result in underfitting.
2. Adding dropout regularization. During training, the nodes of a particular layer may always become influenced only by the output of a particular node in the previous layer, causing overfitting. Dropout regularization is a technique that randomly drops a number of nodes in a neural network during training as a way of adding randomization and preventing nodes from becoming dependent on one another. Adding dropout regularization can reduce overfitting and also improve the performance of the model.

Task:

1. Tweak the variable `num_epochs` above and restart and rerun all of the cells above. Evaluate the performance of the model on the training data and the validation data.
2. Add Keras `Dropout` layers after one or all hidden layers. Add the following line of code after you add a hidden layer to your model object: `nn_model.add(keras.layers.Dropout(.25))`. The parameter `.25` is the fraction of the nodes to drop. You can experiment with this value as well. Restart and rerun all of the cells above. Evaluate the performance of the model on the training data and the validation data.

Analysis: In the cell below, specify the different approaches you used to reduce overfitting and summarize which configuration led to the best generalization performance.

Did changing the number of epochs prevent overfitting? Which value of `num_epochs` yielded the closest training and validation accuracy score?

Did adding dropout layers prevent overfitting? How so? Did it also improve the accuracy score when evaluating the validation data? How many dropout layers did you add and which fraction of nodes did you drop?

Record your findings in the cell below.

1.7.2 Evaluate the Performance on the Test Set

Now that you have improved the model, let's evaluate its performance on our test data and compare the results.

Task: In the code cell below, call the `evaluate()` method on the model object `nn_model`. Specify `X_test_tfidf` and `y_test` as arguments. You must convert `X_test_tfidf` to a NumPy array using the `toarray()` method.

Note: The `evaluate()` method returns a list containing two values. The first value is the loss and the second value is the accuracy score.

[17]:
```
# loss, accuracy = # YOUR CODE HERE

# SOLUTION
loss, accuracy = nn_model.evaluate(X_test_tfidf.toarray(), y_test)
print('Loss: ', str(loss) , 'Accuracy: ', str(accuracy))
```

```
16/16 [==============================] - 0s 4ms/step - loss: 0.5539 - accuracy:
0.8178
Loss:  0.5539271831512451 Accuracy:  0.8178137540817261
```

1.7.3 Make Predictions on the Test Set

Now that we have our best performing model that can generalize to new, previously unseen data, let us make predictions using our test data.

In the cell below, we will make a prediction on our test set and receive probability predictions for every example in the test set (these values will be between 0.0 and 1.0). We will then inspect the results for the first 20 examples — We will apply a threshold to determine the predicted class for every example; we will use a threshold of 0.5. This means that if the probability is greater than 0.5, we will assume the book review is good. We will then print the actual class labels contained in `y_test` to see if our model is producing accurate predictions.

Task: In the code cell below, do the following:

1. Call the `predict()` method on the model object `nn_model`. Specify `X_test_tfidf` as an argument. You must convert `X_test_tfidf` to a NumPy array using the `toarray()` method. Save the results to the array `probability_predictions`.
2. Loop through the first 20 items in `probability_predictions`. These correspond to the predicted probabilities for the first 20 examples in our test set. For every item, check if the probability is greater than 0.5. If so, output:

- the probability value in `probability_predictions`
- the corresponding label in `y_test`. Note: convert the series `y_test` using `y_test.to_numpy()` before indexing into it.

Inspect the results. How is our model performing? Is our model properly predicting whether the book reviews are good or bad reviews?

[18]:
```
# Make predictions on the test set

# probability_predictions = # YOUR CODE HERE

# solution
probability_predictions = nn_model.predict(X_test_tfidf.toarray())

print("Predictions for the first 20 examples:")
# YOUR CODE HERE

# solution (solutions may vary)
for i in range(0,20):
    if probability_predictions[i] >= .5:
```

```
        print("probability: {0} Good Review?: {1}\n".
↪format(probability_predictions[i], y_test.to_numpy()[i]))
```

```
Predictions for the first 20 examples:
Probability                      Class
probability: [0.9985532] Good Review?: False

probability: [0.96287084] Good Review?: True

probability: [0.9727469] Good Review?: True

probability: [0.88757145] Good Review?: False

probability: [0.99957955] Good Review?: True

probability: [0.9991995] Good Review?: True

probability: [0.9423085] Good Review?: True

probability: [0.65622073] Good Review?: False

probability: [0.99803245] Good Review?: True

probability: [0.999329] Good Review?: True

probability: [0.9643129] Good Review?: False

probability: [0.99537915] Good Review?: True

probability: [0.78965604] Good Review?: True

probability: [0.7602196] Good Review?: True
```

Let's check a few of the original book review texts to get a further glimpse into how our model is performing.

[19]:
```
print('Review #1:\n')

print(X_test.to_numpy()[11])

goodReview = True if probability_predictions[11] >= .5 else False

print('\nPrediction: Is this a good review? {}\n'.format(goodReview))

print('Actual: Is this a good review? {}\n'.format(y_test.to_numpy()[11]))
```

```
Review #1:
```

```
Thriller and mystery readers like to guess along with our protagonists when
we're reading a story.  We like to look for clues in what we've read to help us
unravel the plot, and find out who did it and why the crime was done.

John Grisham cheats us out of that fun.  The villian of this book isn't
introduced until there less than 75 pages left, which means that when you get
that far, you realize that everything you read before then wasn't valid. To have
some red herrings in a thriller is fine, but to have 300 pages of chases and red
herrings?  That's bad.

Read another thriller instead of this one, one that plays by the rules


Prediction: Is this a good review? True

Actual: Is this a good review? False
```

[20]:
```
print('Review #2:\n')

print(X_test.to_numpy()[24])

goodReview = True if probability_predictions[24] >= .5 else False

print('\nPrediction: Is this a good review? {}\n'.format(goodReview))

print('Actual: Is this a good review? {}\n'.format(y_test.to_numpy()[24]))
```

```
Review #2:

I have read Baldacci's first four novels and have immensely enjoyed all of them.
This one, however, is just awful.  Not only the character's dialogue but even
the story itself is written like a really bad detective movie.  The only thing I
can think of to compare it to is this :  There was a series of Calvin and Hobbes
cartoons where Calvin imagines himself as a private detective and they are
written like the old detective shows, with lame lines like "The gun was loaded,
and so was I".  That is exactly what this book is like, except it goes on for
400 pages.  There isn't a single interesting character in this book, in my
opinion.  You just have to slog your way through the book to get to the end.
It's the Bataan Death March of novels.  I hope this is an aberration - I'll
certainly give him another try since the first four novels that I read were so
good.  But one more stinker like this one and I'll drop his name from my reading
list


Prediction: Is this a good review? False
```

```
Actual: Is this a good review? False
```

[21]:
```
print('Review #3:\n')

print(X_test.to_numpy()[56])

goodReview = True if probability_predictions[56] >= .5 else False

print('\nPrediction: Is this a good review? {}\n'.format(goodReview))

print('Actual: Is this a good review? {}\n'.format(y_test.to_numpy()[56]))
```

```
Review #3:

This commentary has many tremendous insights into the book of Romans.  Romans is
one of the richest resources of truth within the Bible and John Stoot does a
good job of unpacking it.  The book is written from a framework of the reader
having a strong prior understanding of the Scriptures.  It is probably not
considered a highly scholarly work, but it is not for the average reader to pick
up for light reading either


Prediction: Is this a good review? True

Actual: Is this a good review? True
```

[22]:
```
print('Review #4:\n')
print(X_test.to_numpy()[102])

goodReview = True if probability_predictions[102] >= .5 else False

print('\nPrediction: Is this a good review? {}\n'.format(goodReview))

print('Actual: Is this a good review? {}\n'.format(y_test.to_numpy()[102]))
```

```
Review #4:

With astute attention to the details of character, setting and daily life, Susan
Kelly illuminates the ordinary. Pondering love, memory, faith,and responsiblity,
Kelly transforms the everyday into the quintessential. This is a beautifully
rendered story


Prediction: Is this a good review? True
```

```
Actual: Is this a good review? True
```

1.8 Part 8: Analysis

Experiment with the vectorizer and neural network implementation above and compare your results every time you train the network. Pay attention to the time it takes to train the network, and the resulting loss and accuracy on both the training and test data.

Below are some ideas for things you can try:

- Adjust the learning rate.
- Add more hidden layers and/or experiment with different values for the `unit` parameter in the hidden layers to change the number of nodes in the hidden layers.
- Fit your vectorizer using different document frequency values and different n-gram ranges. When creating a `TfidfVectorizer` object, use the parameter `min_df` to specify the minimum 'document frequency' and use `ngram_range=(1,2)` to change the default n-gram range of `(1,1)`.

Record your findings in the cell below.