**Meets Specifications**

I am really impressed with the amount of effort you've put into the project. You deserve applaud for your hardwork!

Finally, Congratulations on completing this project. You are one step closer to finishing your Nanodegree.

Wishing you good luck for all future projects 

## Deploying a Web App

Answer gives a sample review and the resulting predicted sentiment.

Awesome! The model correctly predicts the sentiment of the sample review.

Suggestion: You could have also tried more complex review where the review may begin with one sentiment and then eventually make up for it and then present an opposite sentiment.  
E.g. A review that starts negatively claiming that the movie was not as good as other movies by the director but overall it was excellent and well worth the time and money spent.

The model is deployed and the Lambda / API Gateway integration is complete so that the web app works (make sure to include your modified index.html).

AWS API is included in index.html

Underlying Mechanism is as follows:  
On clicking the Submit button, the web app hits the AWS Lambda API and returns the model's prediction to the web app.

### Overwhelmed with all the AWS lingo? Checkout this [amazing website](https://expeditedsecurity.com/aws-in-plain-english/) that explains all AWS Services in simple terms.

**Use the Model for Testing**

The predict\_fn() method in serve/predict.py has been implemented.

Nicely done! 

The test review has been processed correctly and stored in the test\_data variable.

predict function executed properly and you've correctly processed the test review.

Good job passing the length of the review to predict function.  
[Graphical user interface, text, application

Description automatically generated](https://udacity-reviews-uploads.s3.us-west-2.amazonaws.com/_attachments/296140/1591709884/sagemaker_test.png)

Suggestion:  
Here's an alternate approach to this task -

review\_list, review\_length = convert\_and\_pad(word\_dict, review\_to\_words(test\_review))

test\_data = np.array([np.array([review\_length] + review\_list)])

Answer describes the differences between the RNN model and the XGBoost model and how they perform on the IMDB data.

When choosing an algorithm we must pay close attention to our data. In our case the data consists of sentences where the context between words and the semantics of the overall sentence is very important. In such cases RNNs/LSTMs work better because they are able to generate a hidden state based on the sequence in which words appear. XGBoost cannot do that, therefore RNNs/LSTMs are **comparably** better at performing sentiment analysis

## Deploy a Model for Testing

The trained PyTorch model is successfully deployed.

The RNN model is successfully deployed to ml.m4.xlarge AWS instance.

## Build and Train a PyTorch Model

The RNN is trained using SageMaker's supported PyTorch functionality.

estimator.fit() executed properly which is an indication that you implemented your train() method correctly.

[Timeline

Description automatically generated](https://udacity-reviews-uploads.s3.us-west-2.amazonaws.com/_attachments/296140/1612402474/train.png)

Note - I verified the implemention in train.py too.

The train method is implemented and can be used to train the PyTorch model.

Good job at implementing the train method correctly.

Zeroing out gradients is a very important step that most people forget to implement.  
For remembering the training steps I use the custom acronym: **ZOLS**  
Z -> zero\_grad()  
O -> output (preds)  
L -> loss  
S -> optimizer.step()

You can create your own custom acronym to remember the training steps.

## Preparing and Processing Data

Answer describes how the processing methods are applied to the training and test data sets and what, if any, issues there may be.

Good observation.  
When pre-processing train and test data, we should use the same pre-processing **steps**. This is because the model that is trained is the same model on which we will test the data. Using same processing steps ensures both training and test data have similar representations.

However, it's important to note that we shouldn't accidentally use testing data while building word\_dict in our case. That'll introduce data leakage and skew results.

Notebook displays the five most frequently appearing words.

The 5 five most frequently appearing words are correctly displayed.

'movi', 'film', 'one', 'like', 'time'

The build\_dict method is implemented and constructs a valid word dictionary.

build\_dict constructs a valid dictionary.

Suggestion: Here's a sample code snippet of an alternate implementation using Python's built-in Counter function.

from collections import Counter

def build\_dict(data, vocab\_size = 5000):

# A dict storing the words that appear in the reviews along with how often they occur

word\_count = Counter(np.concatenate(data))

sorted\_words = sorted(word\_count, key=word\_count.get, reverse=True)

word\_dict = {}

for idx, word in enumerate(sorted\_words[:vocab\_size - 2]):

word\_dict[word] = idx + 2

return word\_dict

Answer describes what the pre-processing method does to a review.

You've correctly pointed out the modifications made by the pre-processing method to a review.

### Note: The function gets rid of punctuations from the review using regular expressions.

text = re.sub(r"[^a-zA-Z0-9]", " ", text.lower())

In the above line of code, re.sub replaces all characters that are NOT alphabets or numbers with a space.  
You can read more about re.sub here: <https://docs.python.org/3/library/re.html#re.sub>

## Files Submitted

The submission includes all required files, including notebook, python scripts, and html files.

All files are included in the submission zip  
Project Notebook  
index.html  
train.py  
predict.py