

[Return to Classroom](#)

Deploying a Sentiment Analysis Model

审阅

代码审阅

HISTORY

符合要求

Dear Student,

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 🍀

Some general suggestions

Use of assertions and Logging:

- Consider using [Python assertions](#) for sanity testing - assertions are great for catching bugs. This is especially true of a dynamically type-checked language like Python where a wrong variable type or shape can cause errors at runtime
- Logging is important for long-running applications. Logging done right produces a report that can be analyzed to debug errors and find crucial information. There could be different levels of logging or logging tags that can be used to filter messages most relevant to someone. Messages can be written to the terminal using `print()` or saved to file, for example using the [Logger module](#). Sometimes it's worthwhile to catch and log exceptions during a long-running operation so that the operation itself is not aborted.

Debugging:

- Check out this guide on [debugging in python](#)

Reproducibility:

- Reproducibility is perhaps the biggest issue in machine learning right now. With so many moving parts present in the code (data, hyperparameters, etc) it is imperative that the instructions and code make it easy for anyone to get exactly the same results (just imagine debugging an ML pipeline where the data changes every time and so you cannot get the same result twice).
- Also consider using random seeds to make your data more reproducible.

Optimization and Profiling:

- Monitoring progress and debugging with [Tensorboard](#): This tool can log detailed information about the model, data, hyperparameters, and more. Tensorboard can be used with Pytorch as well.
- Profiling with Pytorch: [Pytorch's profiler](#) can be used to break down profiling information by operations (convolution, pooling, batch norm) and identify performance bottlenecks. The performance traces can be viewed in the browser itself. The profiler is a great tool for quickly comparing GPU vs CPU speedups for example.

Files Submitted

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

Make sure your submission contains:

- The `SageMaker Project.ipynb` file with fully functional code, all code cells executed and displaying output, and all questions answered.
- An HTML or PDF export of the project notebook with the name `report.html` or `report.pdf`.
- The `train` folder with all provided files and the completed `train.py`.
- The `serve` folder with all provided files and the completed `predict.py`.
- The `website` folder with the edited `index.html` file.

All files are included in the submission zip

- ☒ Project Notebook
- ☒ index.html
- ☒ train.py
- ☒ predict.py


Preparing and Processing Data

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

Question: Above we mentioned that `review_to_words` method removes html formatting and allows us to tokenize the words found in a review, for example, converting *entertained* and *entertaining* into *entertain* so that they are treated as though they are the same word. What else, if anything, does this method do to the input?

Answer: After removing html formatting, the method replaces all uppercase letters with lowercase letters, splits the string into words and remove all insignificant words (stopwords).

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>

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

Question: In the cells above we use the `preprocess_data` and `convert_and_pad_data` methods to process both the training and testing set. Why or why not might this be a problem?

Answer: Using the same preprocessing methods for both training and testing data won't cause any problem. Preprocessing method converts the data into a format that can be used in our model, it's an essential step. Note that we used `word_dict` which is created from the training data to process the testing data. It isn't a problem because we want that our model performs well on new data. But considering the production in real life, the words in `word_dict` may change, and the frequency of the words may also change. So keep the `word_dict` unchanged isn't a good idea. Instead, `word_dict` should be updated regularly. Also we need to avoid the leakage problem.

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.

Question: What are the five most frequently appearing (tokenized) words in the training set? Does it makes sense that these words appear frequently in the training set?

Answer: The five most frequently appearing words in our training data are "movi", "film", "one", "like" and "time". It looks reasonable since our training set is about movie reviews. It's normal for words related to movies and one's feelings to appear the most frequently.

```
# TODO: Use this space to determine the five most frequently appearing words in
the training set.
word_dict_list = list(word_dict.items())
print([word_dict_list[i] for i in range(5)])

[('movi', 2), ('film', 3), ('one', 4), ('like', 5), ('time', 6)]
```

The 5 five most frequently appearing words are correctly displayed.

```
[('movi', 2), ('film', 3), ('one', 4), ('like', 5), ('time', 6)]
```

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 `Counter` module.

```
from collections import Counter

def build_dict(data, vocab_size = 5000):

    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
```

Build and Train a PyTorch Model

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

Good job at implementing the train method correctly.

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.

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.

```
Invoking script with the following command:
/usr/bin/python -m train --epochs 10 --hidden_dim 200
Using device cuda.
Get train data loader.
load f
Model loaded with embedding_dim 32, hidden_dim 200, vocab_size 5000.
Epoch: 1, BCELoss: 0.6696704893696065
Epoch: 2, BCELoss: 0.5924909139166072
Epoch: 3, BCELoss: 0.5042590030602047
Epoch: 4, BCELoss: 0.4371729730343332
Epoch: 5, BCELoss: 0.3874118212534457
Epoch: 6, BCELoss: 0.35067193544640834
Epoch: 7, BCELoss: 0.3213530772802781
Epoch: 8, BCELoss: 0.299233760760755
Epoch: 9, BCELoss: 0.3211515454613433
Epoch: 10, BCELoss: 0.2744908490959479
2022-09-29 15:45:27,584 sagemaker-containers INFO      Reporting training SUCCESS
```

```
2022-09-29 15:45:52 Uploading - Uploading generated training model
2022-09-29 15:45:52 Completed - Training job completed
```

Note - I verified the implementation in `train.py` too.

Deploy a Model for Testing

The trained PyTorch model is successfully deployed.

The RNN model is successfully deployed to `m1.p2.xlarge` AWS instance.

Note: `m4` is a general purpose instance primarily used to host webapps that require significant computer while `p2` is a specialized instance with High Performance GPUs which are useful for ML tasks.

Use the Model for Testing

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

Make sure your answer includes:

- The comparison between the two models
- Which model is better for sentiment analysis

Question: How does this model compare to the XGBoost model you created earlier? Why might these two models perform differently on this dataset? Which do you think is better for sentiment analysis?

Answer: I got 0.86396 as the accuracy_score with the XGBoost model. And 0.855 for this RNN LSTM model. The accuracy of the two models is close. But compared to the XGBoost model, this RNN model takes less time to train. In the file model.py, we can also see that it's a simple model with just two layers. I think that it's not difficult to improve the performance of the model by changing some parameters.

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.

While for shorter sequences such as IMDB data, this performance gap may not be significant, with larger datasets you'll see RNNs/LSTMs outperform other models.

The test review has been processed correctly and stored in the `test_data` variable. The `test_data` should contain two variables: `review_len` and `review[500]`.

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

```
test_review_words = review_to_words(test_review)
test_data, test_data_len = convert_and_pad(word_dict, test_review_words)
test_data = np.array([np.array([test_data_len] + test_data)])
```

Good job passing the length of the review to predict function.

The question we now need to answer is, how do we send this review to our model?

Recall in the first section of this notebook we did a bunch of data processing to the IMDb dataset. In particular, we did two specific things to the provided reviews.

- Removed any html tags and stemmed the input
- Encoded the review as a sequence of integers using `word_dict`

In order process the review we will need to repeat these two steps.

TODO: Using the `review_to_words` and `convert_and_pad` methods from section one, convert `test_review` into a numpy array `test_data` suitable to send to our model. Remember that our model expects input of the form `review_length, review[500]`.


Suggestion:

Here's an alternate approach to this task -

```
review = review_to_words(test_review)
review, review_len = convert_and_pad(word_dict, _review)
test_data = pd.concat([pd.DataFrame([review_len]), pd.DataFrame([review])], axis=1)
```

The `predict_fn()` method in `serve/predict.py` has been implemented.

- The predict script should include both the data processing and the prediction.
- The processing should produce two variables: `data_X` and `data_len`.

Nicely done! 

Deploying a Web App

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`).

```
https://aw7z0dc7z7.execute-api.us-east-1.amazonaws.com/prod
```


AWS API is included in `index.html`

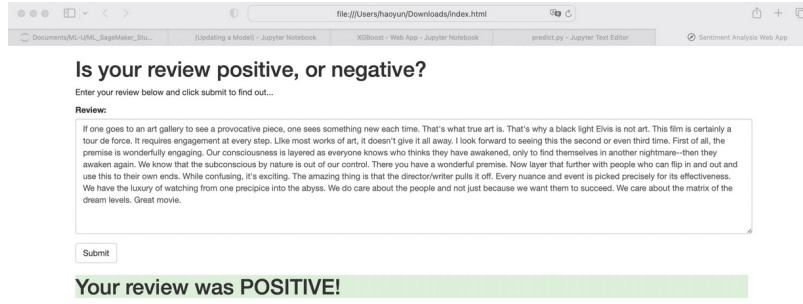
Underlying Mechanism is as follows:

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](#) that explains all AWS Services in simple terms.

The answer includes a screenshot showing a sample review and the prediction.

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



↓ 下载项目

返回 PATH