Deploying a Sentiment Analysis Model

Meets Specifications

Good job with the project! Hope I have answered all the questions to give you more clarity on some of the points.

We use P2 for training as they are GPU instances and m4 are used for deployment and general-purpose deploying web apps which is also crucial to remember while deploying these models in production as the costs are also significant for them.
You can check about the different instances and their usecases here in the AWS documentation - > https://aws.amazon.com/ec2/instance-types/

Do try to use the bigger IMDB set or any other large publicly available datasets and see if you can make a robust sentiment detector model and deploy it as a web app. You will notice RNN improving a bit over XgBoost with more data, more layers and better understanding.

Files Submitted

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

All files are present

Preparing and Processing Data

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

```
download the stopwords
Remove HTML tags (already covered)
converts to lower case
removes non alpha numeric characters
splits into a list of words
removes stopwords
stem the words (already covered)
```

```
text = BeautifulSoup(review, "html.parser").get_text() # Remove HTML tags
    text = re.sub(r"[^a-zA-Z0-9]", " ", text.lower()) # Convert to lower case
    words = text.split() # Split string into words
    words = [w for w in words if w not in stopwords.words("english")] # Remove stopwords
    words = [PorterStemmer().stem(w) for w in words] # stem
```

You actually covered the code description and what it does line by line. Great observation!

Also, if you are interested, do read about Snowball language, its a fun string processing language with many of these algorithms implemented pretty nicely.
http://snowball.tartarus.org/algorithms/porter/stemmer.html

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

Yours was probably the most thorough answer to this question. Max length usually works for fixing the length but as you saw the distribution here is heavily skewed towards reviews with lesser words hence to have that tradeoff of truncating information from those reviews vs getting better speed in training and avoiding sparse 0 values, it made sense to stick to 500.
I would like to add to your explanations.

- One more thing to notice is that we have built our dict specifically from the training set to avoid over-generalizing on every word and avoid data leakage from test to train.
- That being said we can always retrain the model with the new test set data after a certain period of time and include the incoming data after labelling it.

✓ Notebook displays the five most frequently appearing words.

Answer:

movi, film, one, like, time

`movi` is the stemmed version of `movie`

> Does it makes sense that these words appear frequently in the training set?
> yes.

Yep these are movie review neutral words.

✓ The `build_dict` method is implemented and constructs a valid word dictionary.

```
  word_count = {}  # A dict storing the words that appear in the reviews along with how often they occur

    for review in data:
        for word in review:
            word_count[word] = (word_count[word] if word in word_count else 0) + 1

    # DONE: Sort the words found in `data` so that sorted_words[0] is the most frequently appearing word and
    #       sorted_words[-1] is the least frequently appearing word.
    sorted_words = list(map(lambda i: i[0],
                            sorted(word_count.items(), key=lambda i: i[1], reverse=True)
                            ))
```

This does the work.
Tip : You can use list comprehension and python's Counter class to make the code compact and more pythonic (code readability might be bit bad)
It would look something like this ->

```
word_count = dict(Counter(word for sentence in data for word in sentence))
sorted_words = sorted(word_count, key=word_count.get, reverse=True)
```

Build and Train a PyTorch Model

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

We are doing this subset check of training just so that we don't waste any time debugging on our cloud resources for the full dataset

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

```
estimator = PyTorch(entry_point="train.py",
                    source_dir="train",
                    base_job_name=model_name,
                    role=role,
                    framework_version='0.4.0',
                    train_instance_count=1,
                    train_instance_type=TRAIN_INSTANCE,
                    hyperparameters={
                        'epochs': 10,
                        'hidden_dim': 200,
                    })
```

Interesting choice of ml.m5.large. The time taken to train is too long (Training seconds: 11190)
If we used p2 instance for training as it contains gpu, training our model would be much faster (takes about 280s ~ 40x faster)

But trained well on Sagemaker. Could have used less hidden_dim as 200 is a bit of an overkill. It should ideally be in 1x-2x of the embedding dimensions after that it just becomes not necessary most of the time.
Do look at the pytorch class as it also has some IAM and other parameters as well for more control.
https://sagemaker.readthedocs.io/en/stable/sagemaker.pytorch.html

Deploy a Model for Testing

✓ The trained PyTorch model is successfully deployed.

```
predictor = estimator.deploy(
    initial_instance_count=1,
    instance_type=DEPLOY_INSTANCE,
    endpoint_name=endpoint_name
)
predictor = PyTorchPredictor(endpoint_name=endpoint_name)
```

Tip : Do keep an eye on instance type as sometimes we don't have permission to create specific instances and need to get them first from AWS.
m4/m5 are general purpose instance primarily used to host webapps with decent computation.

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.

Here we are using shallow RNNs (max 2 layers) and also the data is not that big (50k reviews). These are some o the reasons why RNN and Xgboost nearly perform similar here. XGBoost works really well with sparse/less data as well as at scale but can't compare to a sequence model like RNN if you want to capture more detailed patterns like context and remember from memory (LSTM)

There are times when boosting methods outperform NNs and viceversa as well.
A good thread to read -> https://datascience.stackexchange.com/questions/2504/deep-learning-vs-gradient-boosting-when-to-use-what

✓ The test review has been processed correctly and stored in the `test_data` variable.

```
test_data, test_data_length = convert_and_pad(word_dict=word_dict, sentence=test_review)
test_data = [test_data.insert(0, test_data_length)]
```

Tip: You can also use the np.hstack + reshape combo or pd.concat to append the two arrays into one

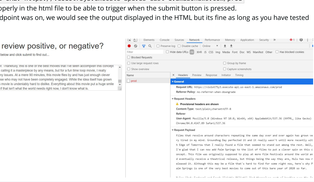
✓ The `predict_fn()` method in `serve/predict.py` has been implemented.

Do use torch.no_grad() before calling model(data) to avoid model to calculate backpropagation as we are only interested in inference here after putting model in eval mode.

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`).

`Request URL: https://vio1ot75y3.execute-api.us-east-1.amazonaws.com/prod`
Used properly in the html file to be able to trigger when the submit button is pressed.
If the endpoint was on, we would see the output displayed in the HTML but its fine as long as you have tested it.



✓ Answer gives a sample review and the resulting predicted sentiment.

Do try to test on neutral reviews or sentences containing double negatives and see what output is generated in XGboost model and RNN model.

DOWNLOAD PROJECT

RETURN TO PATH