UDACITY

‹ Return to Classroom

DISCUSS ON STUDENT HUB

Sentiment Analysis with Neural Networks

REVIEW

HISTORY

Requires Changes

1 specification requires changes

Almost passing . . .

Overall, you are doing Excellent!! You are almost there.

- You have forgotten to calculate and display **validation accuracy** in your training implementation.

Please read through the comments above, correct the issues and resubmit.

Good Luck with your re-submission AND with your remaining 2 projects.

It has been a pleasure reviewing your project.

As for your question posted in the Student Notes . . .

I think this answer at Udacity's Knowledge Base - https://knowledge.udacity.com/questions/30112 answers things much better than I ever could:

Hi **Hahnsang**,

This error is being caused by your LSTM layer. This project is slightly different from the Sentiment RNN you saw in the lessons, one difference is that in this project the inputs come in shape (seq_length, batch_size) instead of (batch_size, seq_length) as in the lesson. Because of this, your LSTM layer should have the parameter:

```
batch_first=False
```

Another difference is the forward pass. As **Ajit** said you shouldn't be reshaping the lstm_out instead you should obtain the **last output of the sequence**, so if the outputs from your LSTM are shape (seq_length, batch_size, lstm_size) you should do:

```
lstm_out[-1,:,:]
```

instead of:

```
lstm_out.contiguous().view(-1, self.lstm_size)
```

After you make this change you just have to pass the outputs through the fully connected layer and logsoftmax without any reshaping, and then returning the logps and the hidden state from the LSTM.

(edited)

Ricardo D almost 3 years ago

SHOW 3 COMMENTS REPORT

Importing Twits

Print the number of twits in the dataset.

Exactly correct!

- You were asked to print out the number of twits encountered (which required you to properly read in the provided **twits.json** file and then understand how to **count** the number of entries) and your code does that properly.

- Nice job – you got the correct number - there are expected to be: **1548010**.

Length of Data

Now let's look at the number of twits in dataset. Print the number of twits below.

```
"""print out the number of twits"""

# TODO Implement
twits_total = len(twits['data'])
twits_total
```

```
1548010
```

Preprocessing the Data

The function `preprocess` correctly lowercases, removes URLs, removes ticker symbols, removes punctuation, tokenizes, and removes any single character tokens.

All requested preprocessing of the tweets has been implemented correctly

You are correctly:

- Lowercasing
- Removing urls
- Removing ticker symbols
- Removing usernames
- Removing non-letter words
- Removing any tokens that are not at least 2 characters long.

Excellent work!

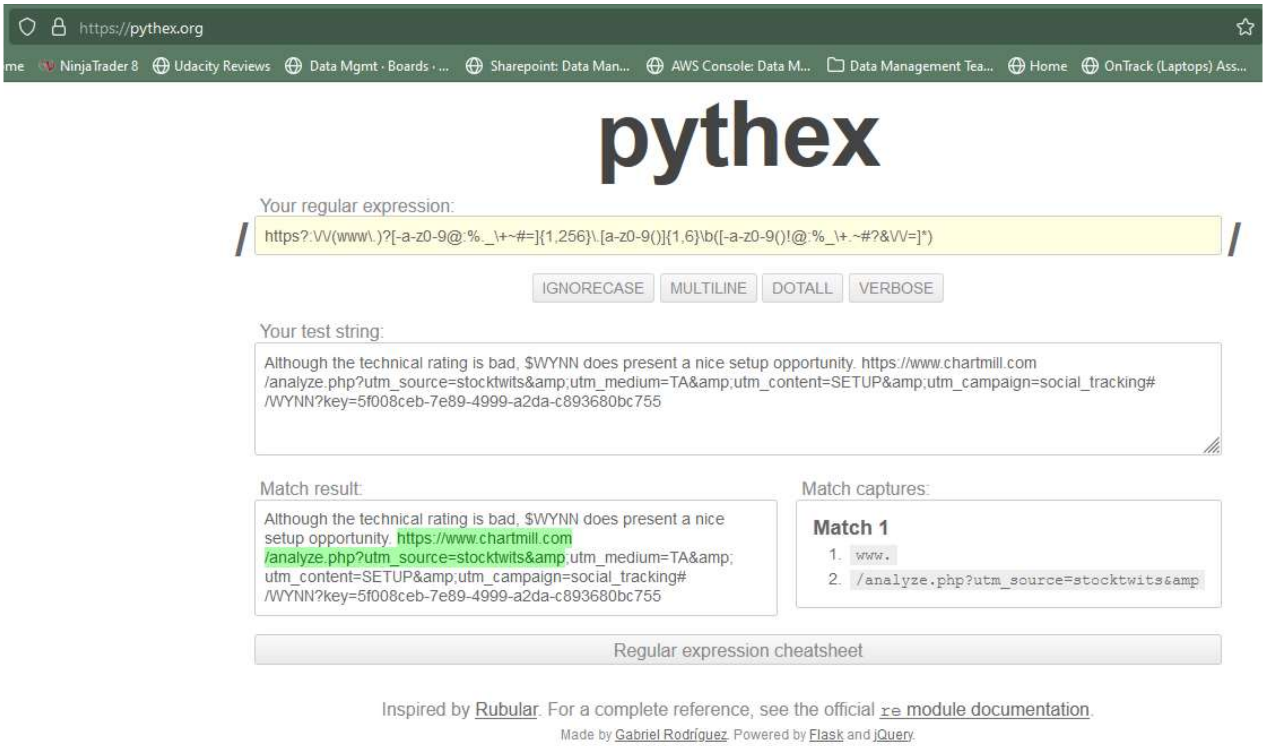
```python
nltk.download('wordnet')

def preprocess(message):
    """
    This function takes a string as input, then performs these operations:
        - lowercase
        - remove URLs
        - remove ticker symbols
        - removes punctuation
        - tokenize by splitting the string on whitespace
        - removes any single character tokens

    Parameters
    ----------
        message : The text message to be preprocessed.

    Returns
    -------
        tokens: The preprocessed text into tokens.
    """
    #TODO: Implement

    # Lowercase the twit message
    text = message.lower()

    # Replace URLs with a space in the message
    # https://ihateregex.io/expr/url/
    text = re.sub(r'https?:\/\/(www\.)?[-a-z0-9@:%._\+~#=]{1,256}\.[a-z0-9()]{1,6}\b([-a-z0-9()!@:%_\+.~#?&\/\/=]*)', ' ',

    # Replace ticker symbols with a space. The ticker symbols are any stock symbol that starts with $.
    text = re.sub(r'\$[a-z0-9]{1,6}', ' ', text)

    # Replace StockTwits usernames with a space. The usernames are any word that starts with @.
    text = re.sub(r'@[a-z0-9@._]+', ' ', text)

    # Replace everything not a letter with a space
    text = re.sub(r'[^a-z]+', ' ', text)

    # Tokenize by splitting the string on whitespace into a list of words
    tokens = text.split(' ')

    # Lemmatize words using the WordNetLemmatizer. You can ignore any word that is not longer than one character.
    wnl = nltk.stem.WordNetLemmatizer()
    tokens = [wnl.lemmatize(token, pos='v') for token in tokens if len(token) > 1]  # root verbs
    tokens = [wnl.lemmatize(token, pos='n') for token in tokens]  # root nouns

    assert type(tokens) == list, 'Tokens should be list'
    return tokens
```

Please Note:

- Your regular expression for urls is quite complex and in fact does not accurately capture ever part of every url.
- While this is not a reason for you to not pass this section, I figured you'd want to be aware of it . . .
- Shown below is one message for which your regex doesn't function properly. The tokens highlighted in red should have been removed:

```
Message: [69]
Although the technical rating is bad, $WYNN does present a nice setup opportunity. https://www.chartmill.com/analyze.php?
utm_source=stocktwits&amp;utm_medium=TA&amp;utm_content=SETUP&amp;utm_campaign=social_tracking#/WYNN?key=5f008ceb-7e89-49
99-a2da-c893680bc755
--------------------------------------------------------------------------------
['although', 'the', 'technical', 'rat', 'be', 'bad', 'do', 'present', 'nice', 'setup', 'opportunity', 'utm', 'medium', 't
a', 'amp', 'utm', 'content', 'setup', 'amp', 'utm', 'campaign', 'social', 'track', 'wynn', 'key', 'ceb', 'da', 'bc']
```

Using a regex tester like the one at https://pythex.org helps us see:



- Also **note**: in this case we are not needing to **validate** a url, just locate them so we can ignore them.
- Since that is the case, a much simpler regex will do the trick:

```
http\S*
```

Inspired by Rubular. For a complete reference, see the official `re` module documentation.
Made by Gabriel Rodríguez. Powered by Flask and jQuery.

Preprocess all the twits into the `tokenized` variable.

Implemented correctly and the first few tokenized elements look appropriate!

- You are using your **preprocess** function and applying it to each message.
- Nice use of a list comprehension!

Preprocess All the Twits

Now we can preprocess each of the twits in our dataset. Apply the function `preprocess` to all the twit messages.

```
# TODO Implement
tokenized = [preprocess(message) for message in messages]
```

Create a bag of words using the tokenized data.

You have properly created a bag of words as you've been taught!

- You have used the **Counter** class properly to create a bag of words from all of the tokens!

Bag of Words

Now with all of our messages tokenized, we want to create a vocabulary and count up how often each word appears in our entire corpus. Use the `Counter` function to count up all the tokens.

```
from collections import Counter


"""
Create a vocabulary by using Bag of words
"""

# TODO: Implement
bow = Counter([word for message in tokenized for word in message])
```

```
bow.most_common(10)
```

```
[('be', 582331),
 ('the', 398861),
 ('to', 379769),
 ('amp', 295583),
 ('for', 273575),
 ('utm', 270812),
 ('on', 241828),
 ('of', 211358),
 ('and', 208609),
 ('in', 205362)]
```

Remove most common and rare words by defining the following variables: `freqs`, `low_cotoff`, `high_cutoff`, `K_most_common`.

You have removed the most common and rare words properly and left a good amount of data for your model to train with!!

- Good Choice for **high_cutoff**
- Good Choice for **low_cutoff**
- You are properly creating **K_most_common**
- You are properly creating **freqs**
- You are removing the rare and most common words in your filtered words list

Well done!

```
"""
Set the following variables:
    freqs
    low_cutoff
    high_cutoff
    K_most_common
"""

# TODO Implement

# Dictionary that contains the Frequency of words appearing in messages.
# The key is the token and the value is the frequency of that word in the corpus.
freqs = {word: count/twits_total for word, count in bow.items()}
print(f'vocabulary: {len(freqs)}')

# Float that is the frequency cutoff. Drop words with a frequency that is lower or equal to this number.
low_cutoff = 0.000002

# Integer that is the cut off for most common words. Drop words that are the `high_cutoff` most common words.
high_cutoff = 20

# The k most common words in the corpus. Use `high_cutoff` as the k.
K_most_common = [word for word, _ in bow.most_common(high_cutoff)]
print(K_most_common)

filtered_words = [word for word in freqs if (freqs[word] > low_cutoff and word not in K_most_common)]
print(f'filtered words: {len(filtered_words)}')
```

```
vocabulary: 92727
['be', 'the', 'to', 'amp', 'for', 'utm', 'on', 'of', 'and', 'in', 'this', 'it', 'at', 'will', 'up', 'buy', 'report', 'go',
'have', 'metric']
filtered words: 27474
```

```
freqs = {word: count / twits_total for word, count in bow.items()}
max_freq = 0.0
min_freq = 1.0
target_low_cutoff=0.000002
max_word = ''
min_word = ''
remove_word_count = 0
for word, freq in freqs.items():
    if freq <= target_low_cutoff:
        remove_word_count += 1
    if freq > max_freq:
        max_freq = freq
        max_word = word
    if freq < min_freq:
        min_freq = freq
        min_word = word
print(f"The word with Maximum Frequency is: '{max_word}' - Frequency: {max_freq}")
print(f"The word with Minimum Frequency is: '{min_word}' - Frequency: {min_freq}")
print(f" With a low cutoff of {target_low_cutoff}, we would remove {remove_word_count} words out of {len(freqs)}")
print(f" That is {100 * (remove_word_count / (len(freqs)))}% of our data . . .")

```

```
The word with Maximum Frequency is: 'be' - Frequency: 0.3761803864316122
The word with Minimum Frequency is: 'raisinf' - Frequency: 6.459906589750712e-07
 With a low cutoff of 2e-06, we would remove 65233 words out of 92727
 That is 70.34952063584501% of our data . . .
```

Defining the variables : 'vacab', 'id2vocab' and 'filtered' correctly.

vocab, id2vocab, and filtered are all properly created – Bravo!!

- you have properly created **vocab** as a dictionary where the key is the **word** and the value is the **id** of the word.
- you have properly created **id2vocab** as a dictionary where the key is the **id** of the word and the value is the **word**.
- you have properly created **filtered** as a list of **only** the words in vocab from the original **tokenized list**.

Additionally, if you take any word and call id2vocab[vocab[word]] you get back the original word.

- Well done!

Updating Vocabulary by Removing Filtered Words

Let's creat three variables that will help with our vocabulary.

```
"""
Set the following variables:
    vocab
    id2vocab
    filtered
"""

#TODO Implement

# A dictionary for the `filtered_words`. The key is the word and value is an id that represents the word.
vocab = {word: index for index, word in enumerate(filtered_words)}

# Reverse of the `vocab` dictionary. The key is word id and value is the word.
id2vocab = {index: word for word, index in vocab.items()}

# tokenized with the words not in `filtered_words` removed.
filtered = [[word for word in message if word in vocab] for message in tokenized]

assert set(vocab.keys()) == set(id2vocab.values()), 'Check vocab and id2vocab dictionaries'
```

```
id2vocab[vocab['great']]
```

```
'great'
```

Neural Network

The init function correctly initializes the following parameters: `self.vocab_size`, `self.embed_size`, `self.lstm_size`, `self.lstm_layers`, `self.dropout`, `self.embedding`, `self.lstm`, and `self.fc`.

Nicely done:

- you are correctly initializing **self.vocab_size**
- you are correctly initializing **self.embed_size**
- you are correctly initializing **self.lstm_size**

- you are correctly initializing **self.lstm_layers**
- you are correctly initializing **self.dropout**
- you are correctly initializing **self.embedding**

```
    def __init__(self, vocab_size, embed_size, lstm_size, output_size, lstm_layers=1, dropout=0.1):
        """
        Initialize the model by setting up the layers.

        Parameters
        ----------
            vocab_size : The vocabulary size.
            embed_size : The embedding layer size.
            lstm_size : The LSTM layer size.
            output_size : The output size.
            lstm_layers : The number of LSTM layers.
            dropout : The dropout probability.
        """

        super().__init__()

        self.vocab_size = vocab_size
        self.embed_size = embed_size
        self.lstm_size = lstm_size
        self.output_size = output_size
        self.lstm_layers = lstm_layers

        # TODO Implement

        # Setup embedding layer
        self.embedding = nn.Embedding(vocab_size, embed_size)

        # Setup LSTM layer
        self.lstm = nn.LSTM(embed_size, lstm_size, lstm_layers, dropout=dropout, batch_first=False)

        # Setup dropout layer
        self.dropout = nn.Dropout(dropout)

        # Setup fully connected linear (dense) layer
        self.fc = nn.Linear(lstm_size, output_size)

        # Setup softmax layer
        self.lsoftmax = nn.LogSoftmax(dim=1)

```

The 'init_hidden' function generates a hidden state

Yes! - you are properly generating a hidden_state using the incoming batch_size.

```
    def init_hidden(self, batch_size):
        """
        Initializes hidden state

        Parameters
        ----------
            batch_size : The size of batches.

        Returns
        -------
            hidden_state

        """

        # TODO Implement

        # Create two new tensors with sizes lstm_layers x batch_size x lstm_size,
        # initialized to zero, for hidden state and cell state of LSTM
        weight = next(self.parameters()).data

        hidden_state = (weight.new(self.lstm_layers, batch_size, self.lstm_size).zero_(),
                        weight.new(self.lstm_layers, batch_size, self.lstm_size).zero_())

        return hidden_state

```

The 'forward' function performs a forward pass of the model the parameter input using the hidden state.

Excellent! - you are performing a forward pass of the model on the incoming nn_input and using the incoming hidden_state.

- Using an embedding layer helps because there are a large number of words in the vocabulary!

```python
    def forward(self, nn_input, hidden_state):
        """
        Perform a forward pass of our model on nn_input.

        Parameters
        ----------
            nn_input : The batch of input to the NN.
            hidden_state : The LSTM hidden state.

        Returns
        -------
            logps: log softmax output
            hidden_state: The new hidden state.

        """

        # TODO Implement
        #batch_size = nn_input.size(0)  # since batch_first param is set to True

        # Pass thru embedding layer
        embeds = self.embedding(nn_input)
        #print(f'embeds shape: {embeds.shape}')

        # Pass thru LSTM layer
        lstm_out, hidden_state = self.lstm(embeds, hidden_state)
        #print(f'lstm_out shape: {lstm_out.shape}')
        #print(f'hidden_state [0] shape: {hidden_state[0].shape}')

        # Stack up LSTM outputs
        #lstm_stack = lstm_out.contiguous().view(-1, self.lstm_size)
        lstm_stack = lstm_out[-1, :, :]
        #print(f'lstm_stack shape: {lstm_stack.shape}')

        # Pass thru dropout layer
        out = self.dropout(lstm_stack)

        # Pass thru dense layer
        out = self.fc(out)
        #print(f'out shape: {out.shape}')

        # Pass thru log softmax layer
        log_ps = self.lsoftmax(out)
        #print(f'log_ps shape: {log_ps.shape}')

        # Reshape to be batch_size first
        #log_ps = log_ps.view(batch_size, -1)

        # Get last batch of output
        #log_ps = log_ps[:, -1]

        return log_ps, hidden_state
```

Training

Correctly split the data into `train_features`, `valid_features`, `train_labels`, and `valid_labels`.

Excellent!

- You have created a nice **80** / **20** split of training and validation data!

```python
"""
Split data into training and validation datasets. Use an appropriate split size.
The features are the `token_ids` and the labels are the `sentiments`.
"""

# TODO Implement

split_frac = 0.8
split_pos = int(len(token_ids) * split_frac)

train_features, valid_features = token_ids[:split_pos], token_ids[split_pos:]
train_labels, valid_labels = sentiments[:split_pos], sentiments[split_pos:]
```

```python
print("\t\t\tFeature Sizes:")
print("Train size: \t\t{}".format(len(train_features)),
      "\nValidation size: \t{}".format(len(valid_features)))

```

```
                        Feature Sizes:
Train size:             823349
Validation size:        205838
```

Train your model with dropout and clip the gradient. Print out the training progress with the loss and accuracy.

Almost there!

- You are calling **optimizer.zerograd()** properly.
- You are getting output from the model.
- You are using **criterion** to calculate the loss and calling **backward()** to perform backpropagation.
- You are using **clip_grad_norm** to prevent exploding gradient problem.
- You are calculating and printing **training loss and validation loss for every 100 steps**

Excellent job so far . . .

However:

- You were asked to calculate and display validation accuracy as well for every 100 steps

You may find this post at Udacity's Knowledge Base https://knowledge.udacity.com/questions/30981 quite helpful!!

```python
"""
Train your model with dropout. Make sure to clip your gradients.
Print the training loss, validation loss, and validation accuracy for every 100 steps.
"""

# Loss and optimazation
learning_rate = 0.001

criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Training parameters
batch_size = 1024
sequence_length = 100

epochs = 3
clip = 5 # gradient clipping

# Printing parameters
print_every = 100

# Turn on training mode
model.train()

# Epoch loop
for e in range(epochs):
    steps = 0
    print('Starting epoch {}/{}'.format(e + 1, epochs))

    # Batch loop
    for text_batch, labels in dataloader(train_features, train_labels, sequence_length=sequence_length, batch_size=batch_s
        # TODO Implement: Train Model
        steps += 1

        # Initialize hidden state
        hidden = model.init_hidden(labels.shape[0])  # at the last iteration of the epoch, rows of the batch will be left-

        # Set Device
        text_batch, labels = text_batch.to(device), labels.to(device)
        for each in hidden:
            each.to(device)

        # Zero accumulated gradients
        model.zero_grad()

        # Get the output from the model
        logps, hidden = model(text_batch, hidden)

        # calculate the loss and perform backprop
        loss = criterion(logps.squeeze(), labels)
        loss.backward()

        # `clip_grad_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.
        nn.utils.clip_grad_norm_(model.parameters(), clip)
        optimizer.step()

        # Loss stats
        if steps % print_every == 0:
            model.eval()

            # TODO Implement: Print metrics
            valid_losses = []

            for text_batch, labels in dataloader(valid_features, valid_labels, sequence_length=sequence_length, batch_size
                valid_hidden = model.init_hidden(labels.shape[0])  # at the last iteration of the epoch, rows of the batch

                text_batch, labels = text_batch.to(device), labels.to(device)
                for each in valid_hidden:
                    each.to(device)

                logps, valid_hidden = model(text_batch, valid_hidden)
                valid_loss = criterion(logps.squeeze(), labels)

                valid_losses.append(valid_loss.item())

            print("Epoch: {}/{}...".format(e + 1, epochs),
                  "Step: {}...".format(steps),
                  "Loss: {:.6f}...".format(loss.item()),
                  "Valid Loss: {:.6f}".format(np.mean(valid_losses)))

            model.train()
```

```
Starting epoch 1/3
Epoch: 1/3... Step: 100... Loss: 0.942924... Valid Loss: 1.016169
Epoch: 1/3... Step: 200... Loss: 0.886049... Valid Loss: 0.934194
Epoch: 1/3... Step: 300... Loss: 0.867592... Valid Loss: 0.902369
Epoch: 1/3... Step: 400... Loss: 0.838153... Valid Loss: 0.883949
Epoch: 1/3... Step: 500... Loss: 0.853582... Valid Loss: 0.865797
Epoch: 1/3... Step: 600... Loss: 0.813141... Valid Loss: 0.856329
Epoch: 1/3... Step: 700... Loss: 0.825543... Valid Loss: 0.845845
Epoch: 1/3... Step: 800... Loss: 0.754917... Valid Loss: 0.838022
Starting epoch 2/3
Epoch: 2/3... Step: 100... Loss: 0.753580... Valid Loss: 0.839457
Epoch: 2/3... Step: 200... Loss: 0.776417... Valid Loss: 0.836485
Epoch: 2/3... Step: 300... Loss: 0.731900... Valid Loss: 0.833061
Epoch: 2/3... Step: 400... Loss: 0.791676... Valid Loss: 0.826225
Epoch: 2/3... Step: 500... Loss: 0.812037... Valid Loss: 0.827216
Epoch: 2/3... Step: 600... Loss: 0.756848... Valid Loss: 0.818247
Epoch: 2/3... Step: 700... Loss: 0.709338... Valid Loss: 0.816720
Epoch: 2/3... Step: 800... Loss: 0.674014... Valid Loss: 0.816762
Starting epoch 3/3
Epoch: 3/3... Step: 100... Loss: 0.686234... Valid Loss: 0.833953
Epoch: 3/3... Step: 200... Loss: 0.727408... Valid Loss: 0.838917
Epoch: 3/3... Step: 300... Loss: 0.739165... Valid Loss: 0.836471
Epoch: 3/3... Step: 500... Loss: 0.718194... Valid Loss: 0.830025
Epoch: 3/3... Step: 600... Loss: 0.711602... Valid Loss: 0.833537
Epoch: 3/3... Step: 700... Loss: 0.768566... Valid Loss: 0.831728
Epoch: 3/3... Step: 800... Loss: 0.742700... Valid Loss: 0.827539
```

Making Predictions

The `predict` function correctly prints out the prediction vector from the trained model.

You have properly displayed an accurately created prediction vector from the trained model.

- Well done!

```python
def predict(text, model, vocab):
    """
    Make a prediction on a single sentence.

    Parameters
    ----------
        text : The string to make a prediction on.
        model : The model to use for making the prediction.
        vocab : Dictionary for word to word ids. The key is the word and the value is the word id.

    Returns
    -------
        pred : Prediction vector
    """

    # TODO Implement

    tokens = preprocess(text)

    # Filter non-vocab words
    tokens = [token for token in tokens if token in vocab]

    # Convert words to ids
    tokens =  [vocab[token] for token in tokens]

    # Adding a batch dimension
    text_input = torch.from_numpy(np.asarray(torch.LongTensor(tokens).view(-1, 1)))

    # Get the NN output
    hidden = model.init_hidden(1)
    logps, _ = model.forward(text_input, hidden)

    # Take the exponent of the NN output to get a range of 0 to 1 for each label.
    pred = torch.exp(logps)

    return pred
```

```python
text = "Google is working on self driving cars, I'm bullish on $goog"

model.eval()
model.to("cpu")

predict(text, model, vocab)
```

```
tensor([[ 0.0000,  0.0071,  0.0065,  0.8282,  0.1582]])
```

Answer what the prediction of the model is and the uncertainty of the prediction.

Excellent Answer based on your displayed prediction vector!

- You have indicated which label the model is most confident with
- You have indicated the **probability** of that label
- You have indicated the **uncertaintly**

Well Done

```python
text = "Google is working on self driving cars, I'm bullish on $goog"

model.eval()
model.to("cpu")

predict(text, model, vocab)
```

```
tensor([[ 0.0000,  0.0071,  0.0065,  0.8282,  0.1582]])
```

Questions: What is the prediction of the model? What is the uncertainty of the prediction?

TODO: Answer Question

The prediction of the model gave highest positive sentiment on the twit 0.8282 (83%). Models always have uncertainty, in this case, this model has 17% uncertainty.

RESUBMIT

DOWNLOAD PROJECT

Best practices for your project resubmission

Ben shares 5 helpful tips to get you through revising and resubmitting your project.

Watch Video (3:01)

RETURN TO PATH