Double-click (or enter) to edit

```
!pip install -U -q PyDrive
```

```
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
```

```
auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)
```

# New Section

```
link='https://drive.google.com/file/d/14s44-xik0NjVtc2R8Gxj6jdI-
```

```
fluff, id = link.split('=')
id = '14s44-xik0NjVtc2R8Gxj6jdI-JE64ysV'
print (id) # Verify that you have everything after '='
```

```
import pandas as pd
downloaded = drive.CreateFile({'id':id})
downloaded.GetContentFile('corona_tweets_01.csv')
dataframe=pd.read_csv("corona_tweets_01.csv", header=None)
```

```
dataframe=pd.DataFrame(dataframe[0])
```

```
dataframe.to_csv("ready_corona_tweets_01.txt", index=False, head
dataframe
```

```

```

```
pip install twarc
```

```
accesstoken='1427751738-lXHLG1ocIx0K83p5OKuqfmfnbtugZn4n1S1GzA7'
accesstokenkey='EcTbXH6bYC2nJEyZGXEakfJiPcFixdS2i4txSYx2cXE0f'
apikey='6iaCc1iamER8NdvOCApHMN7Io'
apisecretkey='qKZ7jnoHFYzhZdgxku1dWAAgTQfUhvMOK40x4L0yy98BRPwYow
```

```
from twarc import Twarc
```

```
t = Twarc(apikey, apisecretkey, accesstoken, accesstokenkey)
```

```
list_tweets=[]
for x,tweet in enumerate(t.hydrate(open('ready_corona_tweets_01.
  if(x==100):
    break;
  list_tweets.append(tweet['full_text'])
```

```
list_tweets
```

```
# Create a function to clean the tweets
import re
```

```
def deEmojify(text):
    regrex_pattern = re.compile(pattern = "["
        u"\U0001F600-\U0001F64F"  # emoticons
        u"\U0001F300-\U0001F5FF"  # symbols & pictographs
        u"\U0001F680-\U0001F6FF"  # transport & map symbols
        u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
                           "]+", flags = re.UNICODE)
    text.encode('ascii', 'ignore').decode('ascii')
    return regrex_pattern.sub(r'',text)

def cleanTxt(text):
 text = re.sub('@[A-Za-z0-9]+', '', text) #Removing @mentions
 text = re.sub('#', '', text) # Removing '#' hash tag
 text = re.sub('RT[\s]+', '', text) # Removing RT
 text = re.sub('https?:\/\/\S+', '', text) # Removing hyperlink
 text = re.sub('\n', '', text) #REmoving Marks
 text = re.sub(':', '', text) #REmoving Marks
 text = re.sub('_', '', text) #REmoving Marks

 text=deEmojify(text)
 return text



list_tweets=list(map(cleanTxt,list_tweets))
```

```
list_tweets_final=[]
for x,tweet in enumerate(t.hydrate(open('ready_corona_tweets_01.
  list_tweets_final.append(tweet['full_text'])
```

```
    14s44-xik0NjVtc2R8Gxj6jdI-JE64ysV
```

```
list_tweets_final=list(map(cleanTxt,list_tweets_final))
```

```
print (len (list_tweets_final))
```

```
data_tweets=pd.DataFrame(list_tweets_final,columns=["Tweets"])
```

```
data_tweets.head()
```

```
    Requirement already satisfied: twarc in /usr/local/lib/pyth
    Requirement already satisfied: requests-oauthlib in /usr/lo
    Requirement already satisfied: python-dateutil in /usr/loca
    Requirement already satisfied: pytest in /usr/local/lib/pyt
    Requirement already satisfied: requests>=2.0.0 in /usr/loca
    Requirement already satisfied: oauthlib>=3.0.0 in /usr/loca
    Requirement already satisfied: six>=1.5 in /usr/local/lib/p
    Requirement already satisfied: attrs>=17.4.0 in /usr/local/
    Requirement already satisfied: atomicwrites>=1.0 in /usr/lo
    Requirement already satisfied: py>=1.5.0 in /usr/local/lib/
    Requirement already satisfied: more-itertools>=4.0.0 in /us
    Requirement already satisfied: setuptools in /usr/local/lib
```

```
data_tweets.to_csv("data_1.csv", index=False, header=None)
```

```
import tweepy
from textblob import TextBlob
from wordcloud import WordCloud
import pandas as pd
import numpy as np
import re
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
```

```
# Create a function to get the subjectivity
def getSubjectivity(text):
   return TextBlob(text).sentiment.subjectivity
```

```
# Create a function to get the polarity
def getPolarity(text):
   return  TextBlob(text).sentiment.polarity


# Create two new columns 'Subjectivity' & 'Polarity'
data_tweets['Subjectivity'] = data_tweets['Tweets'].apply(getSub
data_tweets['Polarity'] = data_tweets['Tweets'].apply(getPolarit

# Show the new dataframe with columns 'Subjectivity' & 'Polarity
data_tweets
```

```
reviews = np.array(data_tweets['Tweets'])
```

```
# word cloud visualization
allWords = ' '.join([twts for twts in data_tweets['Tweets']])
wordCloud = WordCloud(width=500, height=300, random_state=21, ma
```

```
plt.imshow(wordCloud, interpolation="bilinear")
plt.axis('off')
plt.show()
```

```
# Create a function to compute negative (-1), neutral (0) and po
def getAnalysis(score):
  if score < 0:
```

```
    return 'Negative'
  elif score == 0:
    return 'Neutral'
  else:
    return 'Positive'
data_tweets['Analysis'] = data_tweets['Polarity'].apply(getAnaly
# Show the dataframe
data_tweets
```

```
labels=np.array(data_tweets['Analysis'])
```

```
data_tweets['Tweets'].loc[1]
```

```
data_tweets['Analysis'].loc[1]
```

```
# Plotting
plt.figure(figsize=(8,6))
for i in range(0, data_tweets.shape[0]):
  plt.scatter(data_tweets["Polarity"][i], data_tweets["Subjectiv
# plt.scatter(x,y,color)
plt.title('Sentiment Analysis')
plt.xlabel('Polarity')
plt.ylabel('Subjectivity')
plt.show()
```

```
# Print the percentage of positive tweets
ptweets = data_tweets[data_tweets.Analysis == 'Positive']
ptweets = ptweets['Tweets']
```

```
ptweets

round( (ptweets.shape[0] / data_tweets.shape[0]) * 100 , 1)
```

```
# Print the percentage of negative tweets
ntweets = data_tweets[data_tweets.Analysis == 'Negative']
ntweets = ntweets['Tweets']
ntweets

round( (ntweets.shape[0] / data_tweets.shape[0]) * 100, 1)
```

```
# Show the value counts
data_tweets['Analysis'].value_counts()
```

```
# Plotting and visualizing the counts
plt.title('Sentiment Analysis')
plt.xlabel('Sentiment')
```

```
plt.ylabel('Counts')
data_tweets['Analysis'].value_counts().plot(kind = 'bar')
plt.show()
```

```
import numpy as np
import pandas as pd
```

```
#optional
punctuation = '!"#$%&\'()*+,-./:;<=>?[\\]^_`{|}~'
```

```
all_reviews = 'separator'.join(reviews)
all_reviews = all_reviews.lower()
all_text = ''.join([c for c in all_reviews if c not in punctuati

reviews_split = all_text.split('separator')
all_text = ' '.join(reviews_split)
words = all_text.split()

new_reviews = []
for review in reviews_split:
    review = review.split()
    new_text = []
    for word in review:
        if (word[0] != '@') & ('http' not in word) & (~word.isdi
            new_text.append(word)
    new_reviews.append(new_text)
```

```
from collections import Counter

Counter(labels)
```

```
#dictionary mapping to integer
'''encoding'''
counts = Counter(words)
vocab = sorted(counts, key=counts.get, reverse=True)
vocab_to_int = {word: ii for ii, word in enumerate(vocab, 1)}

## use the dict to tokenize each review in reviews_split
## store the tokenized reviews in reviews_ints
reviews_ints = []
for review in new_reviews:
    reviews_ints.append([vocab_to_int[word] for word in review])
```

```
print('Unique words: ', len((vocab_to_int)))  # should ~ 74000+
print()
```

```
# print tokens in first review
print('Tokenized review: \n', reviews_ints[:1])
```

```
#labels encoding
encoded_labels = []
for label in labels:
    if label == 'Neutral':
        encoded_labels.append(1)
    elif label == 'Negative':
        encoded_labels.append(0)
    else:
        encoded_labels.append(1)

encoded_labels = np.asarray(encoded_labels)
print (encoded_labels)
```

```
print(len (reviews_ints))
```

```
arr=np.array(reviews_ints)
print (arr)
```

```
print('Number of reviews before removing outliers: ', len(review

## remove any reviews/labels with zero length from the reviews_i

# get indices of any reviews with length 0
non_zero_idx = [ii for ii, review in enumerate(reviews_ints) if

# remove 0-length reviews and their labels
reviews_ints = [reviews_ints[ii] for ii in non_zero_idx]
encoded_labels = np.array([encoded_labels[ii] for ii in non_zero

print('Number of reviews after removing outliers: ', len(reviews
```

```
for i in range (100):


  print( (reviews_ints[i]))
```

```
def pad_features(reviews_ints, seq_length):
    ''' Return features of review_ints, where each review is pad
        or truncated to the input seq_length.
    '''

    # getting the correct rows x cols shape
    features = np.zeros((len(reviews_ints), seq_length), dtype=i

    # for each review, I grab that review and
    for i, row in enumerate(reviews_ints):
        features[i, -len(row):] = np.array(row)[:seq_length]

    return features
```

```
seq_length = 20

features = pad_features(reviews_ints, seq_length=seq_length)

## test statements - do not change - ##
assert len(features)==len(reviews_ints), "Your features should h
assert len(features[0])==seq_length, "Each feature row should co

# print first 10 values of the first 30 batches
print(features[:30,:10])
```

```
split_frac = 0.8

## split data into training, validation, and test data (features
```

```
split_idx = int(len(features)*split_frac)
train_x, remaining_x = features[:split_idx], features[split_idx:
train_y, remaining_y = encoded_labels[:split_idx], encoded_label

test_idx = int(len(remaining_x)*0.5)
val_x, test_x = remaining_x[:test_idx], remaining_x[test_idx:]
val_y, test_y = remaining_y[:test_idx], remaining_y[test_idx:]

## print out the shapes of your resultant feature data
print("\t\t\tFeature Shapes:")
print("Train set: \t\t{}".format(train_x.shape),
      "\nValidation set: \t{}".format(val_x.shape),
      "\nTest set: \t\t{}".format(test_x.shape))
```

```
import torch
from torch.utils.data import TensorDataset, DataLoader

# create Tensor datasets
train_data = TensorDataset(torch.from_numpy(train_x), torch.from
valid_data = TensorDataset(torch.from_numpy(val_x), torch.from_n
test_data = TensorDataset(torch.from_numpy(test_x), torch.from_n

# dataloaders
batch_size = 50

# make sure the SHUFFLE your training data
train_loader = DataLoader(train_data, shuffle=True, batch_size=b
valid_loader = DataLoader(valid_data, shuffle=True, batch_size=b
test_loader = DataLoader(test_data, shuffle=True, batch_size=bat
```

```
dataiter = iter(train_loader)
sample_x, sample_y = dataiter.next()

print('Sample input size: ', sample_x.size()) # batch_size, seq_
print('Sample input: \n', sample_x)
print()
```

```
print('Sample label size: ', sample_y.size()) # batch_size
print('Sample label: \n', sample_y)
```

```
train_on_gpu=torch.cuda.is_available()

if(train_on_gpu):
    print('Training on GPU.')
else:
    print('No GPU available, training on CPU.')
```

```
import torch.nn as nn

class SentimentRNN(nn.Module):
    """
    The RNN model that will be used to perform Sentiment analysi
    """

    def __init__(self, vocab_size, output_size, embedding_dim, h
        """
        Initialize the model by setting up the layers.
        """
        super(SentimentRNN, self).__init__()

        self.output_size = output_size
        self.n_layers = n_layers
        self.hidden_dim = hidden_dim

        # embedding and LSTM layers
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers,
                            dropout=drop_prob, batch_first=True)
```

```
        # dropout layer
        self.dropout = nn.Dropout(0.3)

        # linear and sigmoid layers
        self.fc = nn.Linear(hidden_dim, output_size)
        self.sig = nn.Sigmoid()


    def forward(self, x, hidden):
        """
        Perform a forward pass of our model on some input and hi
        """
        batch_size = x.size(0)

        # embeddings and lstm_out
        x = x.long()
        embeds = self.embedding(x)
        lstm_out, hidden = self.lstm(embeds, hidden)

        # stack up lstm outputs
        lstm_out = lstm_out.contiguous().view(-1, self.hidden_di

        # dropout and fully-connected layer
        out = self.dropout(lstm_out)
        out = self.fc(out)
        # sigmoid function
        sig_out = self.sig(out)

        # reshape to be batch_size first
        sig_out = sig_out.view(batch_size, -1)
        sig_out = sig_out[:, -1] # get last batch of labels

        # return last sigmoid output and hidden state
        return sig_out, hidden


    def init_hidden(self, batch_size):
        ''' Initializes hidden state '''
        # Create two new tensors with sizes n_layers x batch_siz

        # initialized to zero, for hidden state and cell state o
        weight = next(self.parameters()).data

        if (train_on_gpu):
```

```
            hidden = (weight.new(self.n_layers, batch_size, self
                  weight.new(self.n_layers, batch_size, self.hid
        else:
            hidden = (weight.new(self.n_layers, batch_size, self
                      weight.new(self.n_layers, batch_size, self

        return hidden
```

```
vocab_size = len(vocab_to_int)+1 # +1 for the 0 padding + our wo
output_size = 1
embedding_dim = 400
hidden_dim = 256
n_layers = 2

net = SentimentRNN(vocab_size, output_size, embedding_dim, hidde

print(net)
```

```
lr=0.001

criterion = nn.BCELoss()
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
```

```
epochs = 3 # 3-4 is approx where I noticed the validation loss s

counter = 0
print_every = 100
clip=5 # gradient clipping

# move model to GPU, if available

if(train_on_gpu):
    net.cuda()

net.train()
```

```
# train for some number of epochs
for e in range(epochs):
    # initialize hidden state
    h = net.init_hidden(batch_size)

    # batch loop
    for inputs, labels in train_loader:
        counter += 1

        if(train_on_gpu):
            inputs, labels = inputs.cuda(), labels.cuda()

        # Creating new variables for the hidden state, otherwise
        # we'd backprop through the entire training history
        h = tuple([each.data for each in h])

        # zero accumulated gradients
        net.zero_grad()

        # get the output from the model
        output, h = net(inputs, h)

        # calculate the loss and perform backprop
        loss = criterion(output.squeeze(), labels.float())
        loss.backward()
        # `clip_grad_norm` helps prevent the exploding gradient
        nn.utils.clip_grad_norm_(net.parameters(), clip)
        optimizer.step()

        # loss stats
        if counter % print_every == 0:
            # Get validation loss
            val_h = net.init_hidden(batch_size)
            val_losses = []
            net.eval()
            for inputs, labels in valid_loader:

                # Creating new variables for the hidden state, o
                # we'd backprop through the entire training hist
                val_h = tuple([each.data for each in val_h])

                if(train_on_gpu):
                    inputs, labels = inputs.cuda(), labels.cuda(
```

```
            output, val_h = net(inputs, val_h)
            val_loss = criterion(output.squeeze(), labels.fl


            val_losses.append(val_loss.item())


        net.train()
        print("Epoch: {}/{}...".format(e+1, epochs),
              "Step: {}...".format(counter),
              "Loss: {:.6f}...".format(loss.item()),
              "Val Loss: {:.6f}".format(np.mean(val_losses))
```

```
test_losses = [] # track loss
num_correct = 0

# init hidden state
h = net.init_hidden(batch_size)


net.eval()
# iterate over test data
for inputs, labels in test_loader:
```

```
    # Creating new variables for the hidden state, otherwise
    # we'd backprop through the entire training history
    h = tuple([each.data for each in h])

    if(train_on_gpu):
        inputs, labels = inputs.cuda(), labels.cuda()

    # get predicted outputs
    output, h = net(inputs, h)

    # calculate loss
    test_loss = criterion(output.squeeze(), labels.float())
    test_losses.append(test_loss.item())

    # convert output probabilities to predicted class (0 or 1)
    pred = torch.round(output.squeeze())  # rounds to the neares

    # compare predictions to true label
    correct_tensor = pred.eq(labels.float().view_as(pred))
    correct = np.squeeze(correct_tensor.numpy()) if not train_on
    num_correct += np.sum(correct)


# -- stats! -- ##
# avg test loss
print("Test loss: {:.3f}".format(np.mean(test_losses)))

# accuracy over all test data
test_acc = num_correct/len(test_loader.dataset)
print("Test accuracy: {:.3f}".format(test_acc))
```

```
test_review_neg = 'corona makes me sick'
```

```

```

```
from string import punctuation

def tokenize_review(test_review):
```

```
    test_review = test_review.lower() # lowercase
    # get rid of punctuation
    test_text = ''.join([c for c in test_review if c not in punc

    # splitting by spaces
    test_words = test_text.split()

    # tokens
    test_ints = []
    test_ints.append([vocab_to_int[word] for word in test_words]

    return test_ints

# test code and generate tokenized review
test_ints = tokenize_review(test_review_neg)
print(test_ints)
```

```
seq_length=20
features = pad_features(test_ints, seq_length)

print(features)
```

```
feature_tensor = torch.from_numpy(features)
print(feature_tensor.size())
```

```
def predict(net, test_review, sequence_length=200):

    net.eval()
```

```
    # tokenize review
    test_ints = tokenize_review(test_review)

    # pad tokenized sequence
    seq_length=sequence_length
    features = pad_features(test_ints, seq_length)

    # convert to tensor to pass into your model
    feature_tensor = torch.from_numpy(features)

    batch_size = feature_tensor.size(0)

    # initialize hidden state
    h = net.init_hidden(batch_size)

    if(train_on_gpu):
        feature_tensor = feature_tensor.cuda()

    # get the output from the model
    output, h = net(feature_tensor, h)

    # convert output probabilities to predicted class (0 or 1)
    pred = torch.round(output.squeeze())
    # printing output value, before rounding
    print('Prediction value, pre-rounding: {:.6f}'.format(output

    # print custom response
    if(pred.item()==1):
        print("Positive review detected!")
    else:
        print("Negative review detected.")
```

```
test_review_neg = 'corona is fuck '
pos_review=' there is complete quarantine facility at india'
seq_length=20
```

```
predict(net, test_review_neg, seq_length)
predict(net,pos_review,seq_length)
```

```
plt.plot(test_losses, label='Training loss')
plt.plot(val_losses, label='Validation loss')
ax = plt.gca()
ax.grid(True)
plt.legend()
plt.ylim(ymax=0.8)
```

Double-click (or enter) to edit

```
!pip install -U -q PyDrive
```

```
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
```

```
auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)
```

# New Section

```
link='https://drive.google.com/file/d/14s44-xik0NjVtc2R8Gxj6jdI-
```

```
fluff, id = link.split('=')
id = '14s44-xik0NjVtc2R8Gxj6jdI-JE64ysV'
print (id) # Verify that you have everything after '='
```

```
import pandas as pd
downloaded = drive.CreateFile({'id':id})
downloaded.GetContentFile('corona_tweets_01.csv')
dataframe=pd.read_csv("corona_tweets_01.csv", header=None)
```

```
dataframe=pd.DataFrame(dataframe[0])
```

```
dataframe.to_csv("ready_corona_tweets_01.txt", index=False, head
dataframe
```

```
pip install twarc
```

```
accesstoken='1427751738-lXHLG1ocIx0K83p5OKuqfmfnbtugZn4n1S1GzA7'
accesstokenkey='EcTbXH6bYC2nJEyZGXEakfJiPcFixdS2i4txSYx2cXE0f'
apikey='6iaCc1iamER8NdvOCApHMN7Io'
apisecretkey='qKZ7jnoHFYzhZdgxku1dWAAgTQfUhvMOK40x4L0yy98BRPwYow
```

```
from twarc import Twarc
```

```
t = Twarc(apikey, apisecretkey, accesstoken, accesstokenkey)
```

```
list_tweets=[]
for x,tweet in enumerate(t.hydrate(open('ready_corona_tweets_01.
  if(x==100):
    break;
  list_tweets.append(tweet['full_text'])
```

```
list_tweets
```

```
# Create a function to clean the tweets
import re

def deEmojify(text):
    regrex_pattern = re.compile(pattern = "["
        u"\U0001F600-\U0001F64F"  # emoticons
        u"\U0001F300-\U0001F5FF"  # symbols & pictographs
        u"\U0001F680-\U0001F6FF"  # transport & map symbols
        u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
                           "]+", flags = re.UNICODE)
    text.encode('ascii', 'ignore').decode('ascii')
    return regrex_pattern.sub(r'',text)

def cleanTxt(text):
 text = re.sub('@[A-Za-z0-9]+', '', text) #Removing @mentions
 text = re.sub('#', '', text) # Removing '#' hash tag
 text = re.sub('RT[\s]+', '', text) # Removing RT
 text = re.sub('https?:\/\/\S+', '', text) # Removing hyperlink
 text = re.sub('\n', '', text) #REmoving Marks
 text = re.sub(':', '', text) #REmoving Marks
 text = re.sub('_', '', text) #REmoving Marks

 text=deEmojify(text)
 return text



list_tweets=list(map(cleanTxt,list_tweets))
```

```
list_tweets_final=[]
for x,tweet in enumerate(t.hydrate(open('ready_corona_tweets_01.
  list_tweets_final.append(tweet['full_text'])
```

```
list_tweets_final=list(map(cleanTxt,list_tweets_final))
```

```
print (len (list_tweets_final))
```

```
data_tweets=pd.DataFrame(list_tweets_final,columns=["Tweets"])
```

```
data_tweets.head()
```

```
data_tweets.to_csv("data_1.csv", index=False, header=None)
```

```
import tweepy
from textblob import TextBlob
from wordcloud import WordCloud
import pandas as pd
import numpy as np
import re
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
```

```
# Create a function to get the subjectivity
def getSubjectivity(text):
   return TextBlob(text).sentiment.subjectivity
```

```
# Create a function to get the polarity
def getPolarity(text):
   return  TextBlob(text).sentiment.polarity


# Create two new columns 'Subjectivity' & 'Polarity'
data_tweets['Subjectivity'] = data_tweets['Tweets'].apply(getSub
data_tweets['Polarity'] = data_tweets['Tweets'].apply(getPolarit

# Show the new dataframe with columns 'Subjectivity' & 'Polarity
data_tweets
```

```
reviews = np.array(data_tweets['Tweets'])
```

```
# word cloud visualization
allWords = ' '.join([twts for twts in data_tweets['Tweets']])
wordCloud = WordCloud(width=500, height=300, random_state=21, ma
```

```
plt.imshow(wordCloud, interpolation="bilinear")
plt.axis('off')
plt.show()
```

```
# Create a function to compute negative (-1), neutral (0) and po
def getAnalysis(score):
  if score < 0:
```

```
    return 'Negative'
  elif score == 0:
    return 'Neutral'
  else:
    return 'Positive'
data_tweets['Analysis'] = data_tweets['Polarity'].apply(getAnaly
# Show the dataframe
data_tweets
```

```
labels=np.array(data_tweets['Analysis'])
```

```
data_tweets['Tweets'].loc[1]
```

```
data_tweets['Analysis'].loc[1]
```

```
# Plotting
plt.figure(figsize=(8,6))
for i in range(0, data_tweets.shape[0]):
  plt.scatter(data_tweets["Polarity"][i], data_tweets["Subjectiv
# plt.scatter(x,y,color)
plt.title('Sentiment Analysis')
plt.xlabel('Polarity')
plt.ylabel('Subjectivity')
plt.show()
```

```
# Print the percentage of positive tweets
ptweets = data_tweets[data_tweets.Analysis == 'Positive']
ptweets = ptweets['Tweets']
```

```
ptweets

round( (ptweets.shape[0] / data_tweets.shape[0]) * 100 , 1)
```

```
# Print the percentage of negative tweets
ntweets = data_tweets[data_tweets.Analysis == 'Negative']
ntweets = ntweets['Tweets']
ntweets

round( (ntweets.shape[0] / data_tweets.shape[0]) * 100, 1)
```

```
# Show the value counts
data_tweets['Analysis'].value_counts()
```

```
# Plotting and visualizing the counts
plt.title('Sentiment Analysis')
plt.xlabel('Sentiment')
```

```
plt.ylabel('Counts')
data_tweets['Analysis'].value_counts().plot(kind = 'bar')
plt.show()
```

```
import numpy as np
import pandas as pd
```

```
#optional
punctuation = '!"#$%&\'()*+,-./:;<=>?[\\]^_`{|}~'
```

```
all_reviews = 'separator'.join(reviews)
all_reviews = all_reviews.lower()
all_text = ''.join([c for c in all_reviews if c not in punctuati

reviews_split = all_text.split('separator')
all_text = ' '.join(reviews_split)
words = all_text.split()

new_reviews = []
for review in reviews_split:
    review = review.split()
    new_text = []
    for word in review:
        if (word[0] != '@') & ('http' not in word) & (~word.isdi
            new_text.append(word)
    new_reviews.append(new_text)
```

```
from collections import Counter

Counter(labels)
```

```
#dictionary mapping to integer
'''encoding'''
counts = Counter(words)
vocab = sorted(counts, key=counts.get, reverse=True)
vocab_to_int = {word: ii for ii, word in enumerate(vocab, 1)}

## use the dict to tokenize each review in reviews_split
## store the tokenized reviews in reviews_ints
reviews_ints = []
for review in new_reviews:
    reviews_ints.append([vocab_to_int[word] for word in review])
```

```
print('Unique words: ', len((vocab_to_int)))  # should ~ 74000+
print()
```

```
# print tokens in first review
print('Tokenized review: \n', reviews_ints[:1])
```

```
#labels encoding
encoded_labels = []
for label in labels:
    if label == 'Neutral':
        encoded_labels.append(1)
    elif label == 'Negative':
        encoded_labels.append(0)
    else:
        encoded_labels.append(1)

encoded_labels = np.asarray(encoded_labels)
print (encoded_labels)
```

```
print(len (reviews_ints))
```

```
arr=np.array(reviews_ints)
print (arr)
```

```
print('Number of reviews before removing outliers: ', len(review

## remove any reviews/labels with zero length from the reviews_i

# get indices of any reviews with length 0
non_zero_idx = [ii for ii, review in enumerate(reviews_ints) if

# remove 0-length reviews and their labels
reviews_ints = [reviews_ints[ii] for ii in non_zero_idx]
encoded_labels = np.array([encoded_labels[ii] for ii in non_zero

print('Number of reviews after removing outliers: ', len(reviews
```

```
for i in range (100):


  print( (reviews_ints[i]))
```

```
def pad_features(reviews_ints, seq_length):
    ''' Return features of review_ints, where each review is pad
        or truncated to the input seq_length.
    '''

    # getting the correct rows x cols shape
    features = np.zeros((len(reviews_ints), seq_length), dtype=i

    # for each review, I grab that review and
    for i, row in enumerate(reviews_ints):
        features[i, -len(row):] = np.array(row)[:seq_length]

    return features
```

```
seq_length = 20

features = pad_features(reviews_ints, seq_length=seq_length)

## test statements - do not change - ##
assert len(features)==len(reviews_ints), "Your features should h
assert len(features[0])==seq_length, "Each feature row should co

# print first 10 values of the first 30 batches
print(features[:30,:10])
```

```
split_frac = 0.8

## split data into training, validation, and test data (features
```

```
split_idx = int(len(features)*split_frac)
train_x, remaining_x = features[:split_idx], features[split_idx:
train_y, remaining_y = encoded_labels[:split_idx], encoded_label

test_idx = int(len(remaining_x)*0.5)
val_x, test_x = remaining_x[:test_idx], remaining_x[test_idx:]
val_y, test_y = remaining_y[:test_idx], remaining_y[test_idx:]

## print out the shapes of your resultant feature data
print("\t\t\tFeature Shapes:")
print("Train set: \t\t{}".format(train_x.shape),
      "\nValidation set: \t{}".format(val_x.shape),
      "\nTest set: \t\t{}".format(test_x.shape))
```

```
import torch
from torch.utils.data import TensorDataset, DataLoader

# create Tensor datasets
train_data = TensorDataset(torch.from_numpy(train_x), torch.from
valid_data = TensorDataset(torch.from_numpy(val_x), torch.from_n
test_data = TensorDataset(torch.from_numpy(test_x), torch.from_n

# dataloaders
batch_size = 50

# make sure the SHUFFLE your training data
train_loader = DataLoader(train_data, shuffle=True, batch_size=b
valid_loader = DataLoader(valid_data, shuffle=True, batch_size=b
test_loader = DataLoader(test_data, shuffle=True, batch_size=bat
```

```
dataiter = iter(train_loader)
sample_x, sample_y = dataiter.next()

print('Sample input size: ', sample_x.size()) # batch_size, seq_
print('Sample input: \n', sample_x)
print()
```

```
print('Sample label size: ', sample_y.size()) # batch_size
print('Sample label: \n', sample_y)
```

```
train_on_gpu=torch.cuda.is_available()

if(train_on_gpu):
    print('Training on GPU.')
else:
    print('No GPU available, training on CPU.')
```

```
import torch.nn as nn

class SentimentRNN(nn.Module):
    """
    The RNN model that will be used to perform Sentiment analysi
    """

    def __init__(self, vocab_size, output_size, embedding_dim, h
        """
        Initialize the model by setting up the layers.
        """
        super(SentimentRNN, self).__init__()

        self.output_size = output_size
        self.n_layers = n_layers
        self.hidden_dim = hidden_dim

        # embedding and LSTM layers
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers,
                            dropout=drop_prob, batch_first=True)
```

```
        # dropout layer
        self.dropout = nn.Dropout(0.3)

        # linear and sigmoid layers
        self.fc = nn.Linear(hidden_dim, output_size)
        self.sig = nn.Sigmoid()


    def forward(self, x, hidden):
        """
        Perform a forward pass of our model on some input and hi
        """
        batch_size = x.size(0)

        # embeddings and lstm_out
        x = x.long()
        embeds = self.embedding(x)
        lstm_out, hidden = self.lstm(embeds, hidden)

        # stack up lstm outputs
        lstm_out = lstm_out.contiguous().view(-1, self.hidden_di

        # dropout and fully-connected layer
        out = self.dropout(lstm_out)
        out = self.fc(out)
        # sigmoid function
        sig_out = self.sig(out)

        # reshape to be batch_size first
        sig_out = sig_out.view(batch_size, -1)
        sig_out = sig_out[:, -1] # get last batch of labels

        # return last sigmoid output and hidden state
        return sig_out, hidden


    def init_hidden(self, batch_size):
        ''' Initializes hidden state '''
        # Create two new tensors with sizes n_layers x batch_siz

        # initialized to zero, for hidden state and cell state o
        weight = next(self.parameters()).data

        if (train_on_gpu):
```

```
            hidden = (weight.new(self.n_layers, batch_size, self
                  weight.new(self.n_layers, batch_size, self.hid
        else:
            hidden = (weight.new(self.n_layers, batch_size, self
                      weight.new(self.n_layers, batch_size, self

        return hidden
```

```
vocab_size = len(vocab_to_int)+1 # +1 for the 0 padding + our wo
output_size = 1
embedding_dim = 400
hidden_dim = 256
n_layers = 2

net = SentimentRNN(vocab_size, output_size, embedding_dim, hidde

print(net)
```

```
lr=0.001

criterion = nn.BCELoss()
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
```

```
epochs = 3 # 3-4 is approx where I noticed the validation loss s

counter = 0
print_every = 100
clip=5 # gradient clipping

# move model to GPU, if available

if(train_on_gpu):
    net.cuda()

net.train()
```

```
# train for some number of epochs
for e in range(epochs):
    # initialize hidden state
    h = net.init_hidden(batch_size)

    # batch loop
    for inputs, labels in train_loader:
        counter += 1

        if(train_on_gpu):
            inputs, labels = inputs.cuda(), labels.cuda()

        # Creating new variables for the hidden state, otherwise
        # we'd backprop through the entire training history
        h = tuple([each.data for each in h])

        # zero accumulated gradients
        net.zero_grad()

        # get the output from the model
        output, h = net(inputs, h)

        # calculate the loss and perform backprop
        loss = criterion(output.squeeze(), labels.float())
        loss.backward()
        # `clip_grad_norm` helps prevent the exploding gradient
        nn.utils.clip_grad_norm_(net.parameters(), clip)
        optimizer.step()

        # loss stats
        if counter % print_every == 0:
            # Get validation loss
            val_h = net.init_hidden(batch_size)
            val_losses = []
            net.eval()
            for inputs, labels in valid_loader:

                # Creating new variables for the hidden state, o
                # we'd backprop through the entire training hist
                val_h = tuple([each.data for each in val_h])

                if(train_on_gpu):
                    inputs, labels = inputs.cuda(), labels.cuda(
```

```
            output, val_h = net(inputs, val_h)
            val_loss = criterion(output.squeeze(), labels.fl


            val_losses.append(val_loss.item())


        net.train()
        print("Epoch: {}/{}...".format(e+1, epochs),
              "Step: {}...".format(counter),
              "Loss: {:.6f}...".format(loss.item()),
              "Val Loss: {:.6f}".format(np.mean(val_losses))
```

```
test_losses = [] # track loss
num_correct = 0

# init hidden state
h = net.init_hidden(batch_size)


net.eval()
# iterate over test data
for inputs, labels in test_loader:
```

```
    # Creating new variables for the hidden state, otherwise
    # we'd backprop through the entire training history
    h = tuple([each.data for each in h])

    if(train_on_gpu):
        inputs, labels = inputs.cuda(), labels.cuda()

    # get predicted outputs
    output, h = net(inputs, h)

    # calculate loss
    test_loss = criterion(output.squeeze(), labels.float())
    test_losses.append(test_loss.item())

    # convert output probabilities to predicted class (0 or 1)
    pred = torch.round(output.squeeze())  # rounds to the neares

    # compare predictions to true label
    correct_tensor = pred.eq(labels.float().view_as(pred))
    correct = np.squeeze(correct_tensor.numpy()) if not train_on
    num_correct += np.sum(correct)


# -- stats! -- ##
# avg test loss
print("Test loss: {:.3f}".format(np.mean(test_losses)))

# accuracy over all test data
test_acc = num_correct/len(test_loader.dataset)
print("Test accuracy: {:.3f}".format(test_acc))
```

```
test_review_neg = 'corona makes me sick'
```

```

```

```
from string import punctuation

def tokenize_review(test_review):
```

```
    test_review = test_review.lower() # lowercase
    # get rid of punctuation
    test_text = ''.join([c for c in test_review if c not in punc

    # splitting by spaces
    test_words = test_text.split()

    # tokens
    test_ints = []
    test_ints.append([vocab_to_int[word] for word in test_words]

    return test_ints

# test code and generate tokenized review
test_ints = tokenize_review(test_review_neg)
print(test_ints)
```

```
seq_length=20
features = pad_features(test_ints, seq_length)

print(features)
```

```
feature_tensor = torch.from_numpy(features)
print(feature_tensor.size())
```

```
def predict(net, test_review, sequence_length=200):

    net.eval()
```

```
    # tokenize review
    test_ints = tokenize_review(test_review)

    # pad tokenized sequence
    seq_length=sequence_length
    features = pad_features(test_ints, seq_length)

    # convert to tensor to pass into your model
    feature_tensor = torch.from_numpy(features)

    batch_size = feature_tensor.size(0)

    # initialize hidden state
    h = net.init_hidden(batch_size)

    if(train_on_gpu):
        feature_tensor = feature_tensor.cuda()

    # get the output from the model
    output, h = net(feature_tensor, h)

    # convert output probabilities to predicted class (0 or 1)
    pred = torch.round(output.squeeze())
    # printing output value, before rounding
    print('Prediction value, pre-rounding: {:.6f}'.format(output

    # print custom response
    if(pred.item()==1):
        print("Positive review detected!")
    else:
        print("Negative review detected.")
```

```
test_review_neg = 'corona is fuck '
pos_review=' there is complete quarantine facility at india'
seq_length=20
```

```
predict(net, test_review_neg, seq_length)
predict(net,pos_review,seq_length)
```

```
plt.plot(test_losses, label='Training loss')
plt.plot(val_losses, label='Validation loss')
ax = plt.gca()
ax.grid(True)
plt.legend()
plt.ylim(ymax=0.8)
```