

sentiment_analysis_using_lstm_pytorch

May 6, 2022

In this kernel we will go through a sentiment analysis on imdb dataset using LSTM.

```
[1]: import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import torch
import torch.optim
import torch.nn as nn
from torch.nn import Parameter
from torch import autograd
import torch.nn.functional as F
from torch.autograd import Variable
from nltk.corpus import stopwords
from collections import Counter
import string
import re
import seaborn as sns
from tqdm import tqdm
import matplotlib.pyplot as plt
from torch.utils.data import TensorDataset, DataLoader
from sklearn.model_selection import train_test_split
```

```
[2]: is_cuda = torch.cuda.is_available()

# If we have a GPU available, we'll set our device to GPU. We'll use this ↵
↵ device variable later in our code.
if is_cuda:
    device = torch.device("cuda")
    print("GPU is available")
else:
    device = torch.device("cpu")
    print("GPU not available, CPU used")
```

GPU is available

```
[3]: from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/MyDrive/asr/sentiment2/
```

Drive already mounted at /content/drive; to attempt to forcibly remount, call

```
drive.mount("/content/drive", force_remount=True).  
/content/drive/MyDrive/asr/sentiment2
```

```
[4]: !ls
```

```
core_qnn          __pycache__          small.csv  
'IMDB Dataset.csv' recurrent_models.py  state_dict.pt
```

```
[5]: base_csv = 'small.csv'  
df = pd.read_csv(base_csv)  
df.head()
```

```
[5]:                                     review sentiment  
0  One of the other reviewers has mentioned that ... positive  
1  A wonderful little production. <br /><br />The... positive  
2  I thought this was a wonderful way to spend ti... positive  
3  Basically there's a family where a little boy ... negative  
4  Petter Mattei's "Love in the Time of Money" is... positive
```

0.0.1 Splitting to train and test data

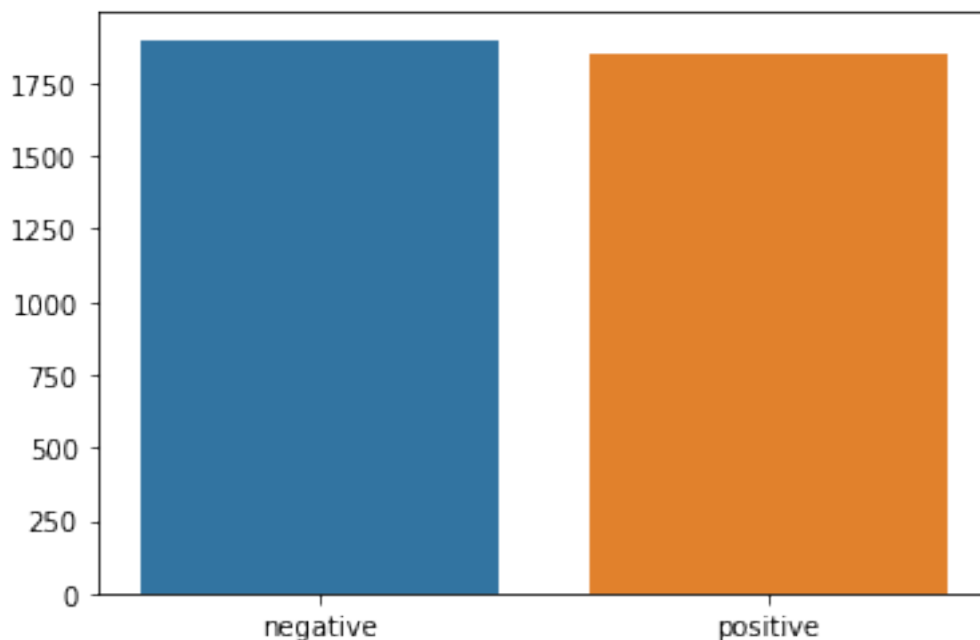
We will split data to train and test initially. Doing this on earlier stage allows to avoid data leakage.

```
[6]: X,y = df['review'].values,df['sentiment'].values  
x_train,x_test,y_train,y_test = train_test_split(X,y,stratify=y)  
print(f'shape of train data is {x_train.shape}')  
print(f'shape of test data is {x_test.shape}')
```

```
shape of train data is (3750,)  
shape of test data is (1250,)
```

0.0.2 Analysing sentiment

```
[7]: dd = pd.Series(y_train).value_counts()  
sns.barplot(x=np.array(['negative','positive']),y=dd.values)  
plt.show()
```



0.0.3 Tokenization

```
[8]: def preprocess_string(s):  
    # Remove all non-word characters (everything except numbers and letters)  
    s = re.sub(r"[^\w\s]", '', s)  
    # Replace all runs of whitespaces with no space  
    s = re.sub(r"\s+", '', s)  
    # replace digits with no space  
    s = re.sub(r"\d", '', s)  
  
    return s  
  
def tokenize(x_train,y_train,x_val,y_val):  
    word_list = []  
  
    stop_words = set(stopwords.words('english'))  
    for sent in x_train:  
        for word in sent.lower().split():  
            word = preprocess_string(word)  
            if word not in stop_words and word != '':  
                word_list.append(word)  
  
    corpus = Counter(word_list)  
    # sorting on the basis of most common words  
    corpus_ = sorted(corpus,key=corpus.get,reverse=True)[:1000]
```

```

# creating a dict
onehot_dict = {w:i+1 for i,w in enumerate(corpus_)}

# tokenize
final_list_train,final_list_test = [],[]
for sent in x_train:
    final_list_train.append([onehot_dict[preprocess_string(word)] for
↪word in sent.lower().split()
                                if preprocess_string(word) in onehot_dict.
↪keys()])
    for sent in x_val:
        final_list_test.append([onehot_dict[preprocess_string(word)] for
↪word in sent.lower().split()
                                if preprocess_string(word) in onehot_dict.
↪keys()])

    encoded_train = [1 if label == 'positive' else 0 for label in y_train]
    encoded_test = [1 if label == 'positive' else 0 for label in y_val]
    return np.array(final_list_train), np.array(encoded_train),np.
↪array(final_list_test), np.array(encoded_test),onehot_dict

```

```

[9]: !pip3 install nltk
import nltk
nltk.download('stopwords')

```

Requirement already satisfied: nltk in /usr/local/lib/python3.7/dist-packages (3.2.5)

Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from nltk) (1.15.0)

[nltk_data] Downloading package stopwords to /root/nltk_data...

[nltk_data] Package stopwords is already up-to-date!

[9]: True

```

[10]: x_train,y_train,x_test,y_test,vocab = tokenize(x_train,y_train,x_test,y_test)

```

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:38:

VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.

```

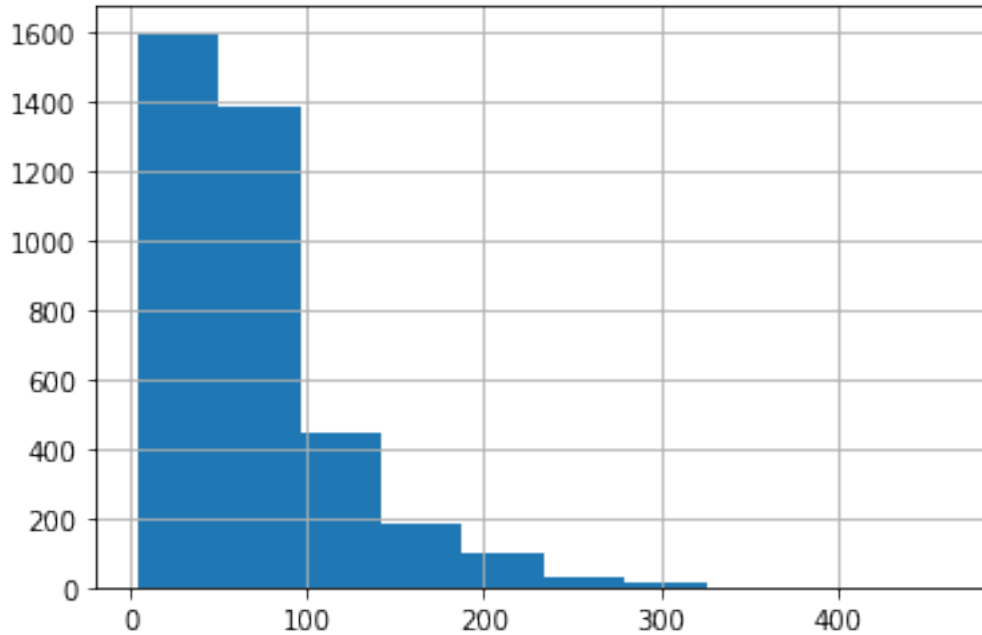
[11]: print(f'Length of vocabulary is {len(vocab)}')

```

Length of vocabulary is 1000

0.0.4 Analysing review length

```
[12]: rev_len = [len(i) for i in x_train]
      pd.Series(rev_len).hist()
      plt.show()
      pd.Series(rev_len).describe()
```



```
[12]: count    3750.000000
      mean      69.567200
      std       48.585543
      min        4.000000
      25%       39.000000
      50%       54.000000
      75%       85.000000
      max      464.000000
      dtype: float64
```

Observations : a) Mean review length = around 69. b) minimum length of reviews is 2.c) There are quite a few reviews that are extremely long, we can manually investigate them to check whether we need to include or exclude them from our analysis.

0.0.5 Padding

Now we will pad each of the sequence to max length

```
[13]: def padding_(sentences, seq_len):
      features = np.zeros((len(sentences), seq_len), dtype=int)
```

```

for ii, review in enumerate(sentences):
    if len(review) != 0:
        features[ii, -len(review):] = np.array(review)[:seq_len]
return features

```

```

[14]: #we have very less number of reviews with length > 500.
#So we will consider only those below it.
x_train_pad = padding_(x_train,500)
x_test_pad = padding_(x_test,500)

```

0.0.6 Batching and loading as tensor

```

[15]: # create Tensor datasets
train_data = TensorDataset(torch.from_numpy(x_train_pad), torch.
    ↪from_numpy(y_train))
valid_data = TensorDataset(torch.from_numpy(x_test_pad), torch.
    ↪from_numpy(y_test))

# dataloaders
batch_size = 50

# make sure to SHUFFLE your data
train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size)
valid_loader = DataLoader(valid_data, shuffle=True, batch_size=batch_size)

```

```

[16]: # obtain one batch of training data
dataiter = iter(train_loader)
sample_x, sample_y = dataiter.next()

print('Sample input size: ', sample_x.size()) # batch_size, seq_length
print('Sample input: \n', sample_x)
print('Sample output: \n', sample_y)

```

Sample input size: torch.Size([50, 500])

Sample input:

```

tensor([[ 0,  0,  0, ..., 495, 863, 656],
        [ 0,  0,  0, ..., 150,  23,   2],
        [ 0,  0,  0, ..., 627, 379,  62],
        ...,
        [ 0,  0,  0, ...,   4, 661, 278],
        [ 0,  0,  0, ..., 944, 676, 177],
        [ 0,  0,  0, ..., 383,   1,  18]])

```

Sample output:

```

tensor([0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1,
        0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0,
        0, 1])

```

We need to add an embedding layer because there are less words in our vocabulary. It is massively

inefficient to one-hot encode that many classes. So, instead of one-hot encoding, we can have an embedding layer and use that layer as a lookup table. You could train an embedding layer using Word2Vec, then load it here. But, it's fine to just make a new layer, using it for only dimensionality reduction, and let the network learn the weights.

0.0.7 Model

```
[17]: import math
from core_qnn.quaternion_layers import *

class LSTM(nn.Module):
    def __init__(self, input_sz: int, hidden_sz: int):
        super().__init__()
        self.input_size = input_sz
        self.hidden_size = hidden_sz

        #i_t
        self.W_i = nn.Linear(input_sz, hidden_sz)
        self.U_i = nn.Linear(hidden_sz, hidden_sz, bias=False)

        #f_t
        self.W_f = nn.Linear(input_sz, hidden_sz)
        self.U_f = nn.Linear(hidden_sz, hidden_sz, bias=False)

        #c_t
        self.W_c = nn.Linear(input_sz, hidden_sz)
        self.U_c = nn.Linear(hidden_sz, hidden_sz, bias=False)

        #o_t
        self.W_o = nn.Linear(input_sz, hidden_sz)
        self.U_o = nn.Linear(hidden_sz, hidden_sz, bias=False)

        self.init_weights()

    def init_weights(self):
        stdv = 1.0 / math.sqrt(self.hidden_size)
        for weight in self.parameters():
            weight.data.uniform_(-stdv, stdv)

    def forward(self,
                x,
                init_states=None):
        """
        assumes x.shape represents (batch_size, sequence_size, input_size)
        """
        bs, seq_sz, _ = x.size()
```

```

hidden_seq = []

if init_states is None:
    h_t, c_t = (
        torch.zeros(bs, self.hidden_size),
        torch.zeros(bs, self.hidden_size),
    )
else:
    h_t, c_t = init_states

for t in range(seq_sz):
    x_t = x[:, t, :]
    i_t = torch.sigmoid(self.W_i(x_t) + self.U_i(h_t))
    f_t = torch.sigmoid(self.W_f(x_t) + self.U_f(h_t))
    g_t = torch.tanh(self.W_c(x_t) + self.U_c(h_t))
    o_t = torch.sigmoid(self.W_o(x_t) + self.U_o(h_t))
    c_t = f_t * c_t + i_t * g_t
    h_t = o_t * torch.tanh(c_t)

    hidden_seq.append(h_t.unsqueeze(0))

#reshape hidden_seq p/ retornar
hidden_seq = torch.cat(hidden_seq, dim=0)
hidden_seq = hidden_seq.transpose(0, 1).contiguous()
return hidden_seq, (h_t, c_t)

```

```

[18]: class SentimentRNN(nn.Module):
    def __init__(self, no_layers, vocab_size, hidden_dim, embedding_dim, drop_prob=0.
    ↪5):
        super(SentimentRNN, self).__init__()

        self.output_dim = output_dim
        self.hidden_dim = hidden_dim

        self.no_layers = no_layers
        self.vocab_size = vocab_size

        # embedding and LSTM layers
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        #lstm
        self.lstm = nn.LSTM(input_size=embedding_dim, hidden_size=self.
        ↪hidden_dim,
                                num_layers=no_layers, batch_first=True)
        # self.lstm = LSTM(embedding_dim, self.hidden_dim)

```



```

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

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

    def forward(self, x, hidden):
        batch_size = x.size(0)
        # embeddings and lstm_out
        embeds = self.embedding(x) # shape: B x S x Feature since batch = 1
        → True
        # print(embeds.shape) #[50, 500, 1000]
        lstm_out, hidden = self.lstm(embeds, hidden)

        lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim)

        # 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_size x hidden_dim,
        # initialized to zero, for hidden state and cell state of LSTM
        h0 = torch.zeros((self.no_layers, batch_size, self.hidden_dim)).to(device)
        c0 = torch.zeros((self.no_layers, batch_size, self.hidden_dim)).to(device)
        hidden = (h0, c0)
        return hidden

```

```

[19]: no_layers = 2
      vocab_size = len(vocab) + 1 #extra 1 for padding

```

```

embedding_dim = 64
output_dim = 1
hidden_dim = 256

model = SentimentRNN(no_layers,vocab_size,hidden_dim,embedding_dim,drop_prob=0.
    ↪5)

#moving to gpu
model.to(device)

print(model)

```

```

SentimentRNN(
  (embedding): Embedding(1001, 64)
  (lstm): LSTM(64, 256, num_layers=2, batch_first=True)
  (dropout): Dropout(p=0.3, inplace=False)
  (fc): Linear(in_features=256, out_features=1, bias=True)
  (sig): Sigmoid()
)

```

0.0.8 Training

```

[20]: # loss and optimization functions
lr=0.001

criterion = nn.BCELoss()

optimizer = torch.optim.Adam(model.parameters(), lr=lr)

# function to predict accuracy
def acc(pred,label):
    pred = torch.round(pred.squeeze())
    return torch.sum(pred == label.squeeze()).item()

```

```

[21]: clip = 5
epochs = 5
valid_loss_min = np.Inf
# train for some number of epochs
epoch_tr_loss,epoch_vl_loss = [],[]
epoch_tr_acc,epoch_vl_acc = [],[]

for epoch in range(epochs):
    train_losses = []
    train_acc = 0.0
    model.train()
    # initialize hidden state

```

```

h = model.init_hidden(batch_size)
for inputs, labels in train_loader:

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

    model.zero_grad()
    output, _ = model(inputs, h)

    # calculate the loss and perform backprop
    loss = criterion(output.squeeze(), labels.float())
    loss.backward()
    train_losses.append(loss.item())
    # calculating accuracy
    accuracy = acc(output, labels)
    train_acc += accuracy
    #`clip_grad_norm` helps prevent the exploding gradient problem in RNNs /
    → LSTMs.
    nn.utils.clip_grad_norm_(model.parameters(), clip)
    optimizer.step()

val_h = model.init_hidden(batch_size)
val_losses = []
val_acc = 0.0
model.eval()
for inputs, labels in valid_loader:
    val_h = tuple([each.data for each in val_h])

    inputs, labels = inputs.to(device), labels.to(device)

    output, val_h = model(inputs, val_h)
    val_loss = criterion(output.squeeze(), labels.float())

    val_losses.append(val_loss.item())

    accuracy = acc(output, labels)
    val_acc += accuracy

epoch_train_loss = np.mean(train_losses)
epoch_val_loss = np.mean(val_losses)
epoch_train_acc = train_acc/len(train_loader.dataset)
epoch_val_acc = val_acc/len(valid_loader.dataset)
epoch_tr_loss.append(epoch_train_loss)

```

```

epoch_vl_loss.append(epoch_val_loss)
epoch_tr_acc.append(epoch_train_acc)
epoch_vl_acc.append(epoch_val_acc)
print(f'Epoch {epoch+1}')
print(f'train_loss : {epoch_train_loss} val_loss : {epoch_val_loss}')
print(f'train_accuracy : {epoch_train_acc*100} val_accuracy : {
→{epoch_val_acc*100}')
    if epoch_val_loss <= valid_loss_min:
        torch.save(model.state_dict(), 'state_dict.pt')
        print('Validation loss decreased ({:.6f} --> {:.6f}). Saving model ...
→'.format(valid_loss_min, epoch_val_loss))
        valid_loss_min = epoch_val_loss
    print(25*'==')

```

Epoch 1

train_loss : 0.6491507577896118 val_loss : 0.6345465528964996

train_accuracy : 60.8 val_accuracy : 66.8

Validation loss decreased (inf --> 0.634547). Saving model ...

=====

Epoch 2

train_loss : 0.5555488741397858 val_loss : 0.5569821882247925

train_accuracy : 72.82666666666667 val_accuracy : 72.39999999999999

Validation loss decreased (0.634547 --> 0.556982). Saving model ...

=====

Epoch 3

train_loss : 0.4759652237097422 val_loss : 0.510241745710373

train_accuracy : 78.0 val_accuracy : 75.92

Validation loss decreased (0.556982 --> 0.510242). Saving model ...

=====

Epoch 4

train_loss : 0.408713006178538 val_loss : 0.5058892726898193

train_accuracy : 81.78666666666666 val_accuracy : 76.32

Validation loss decreased (0.510242 --> 0.505889). Saving model ...

=====

Epoch 5

train_loss : 0.34679873764514924 val_loss : 0.47814624071121214

train_accuracy : 85.44 val_accuracy : 78.8

Validation loss decreased (0.505889 --> 0.478146). Saving model ...

=====

```

[22]: fig = plt.figure(figsize = (20, 6))
plt.subplot(1, 2, 1)
plt.plot(epoch_tr_acc, label='Train Acc')
plt.plot(epoch_vl_acc, label='Validation Acc')
plt.title("Accuracy")
plt.legend()

```

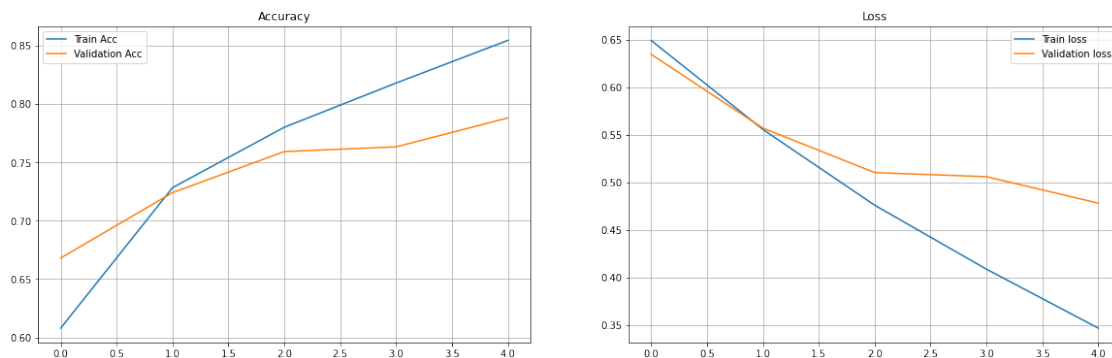
```

plt.grid()

plt.subplot(1, 2, 2)
plt.plot(epoch_tr_loss, label='Train loss')
plt.plot(epoch_vl_loss, label='Validation loss')
plt.title("Loss")
plt.legend()
plt.grid()

plt.show()

```



0.0.9 Inference

```

[23]: def predict_text(text):
        word_seq = np.array([vocab[preprocess_string(word)] for word in text.
        ↪split()
                                if preprocess_string(word) in vocab.keys()])
        word_seq = np.expand_dims(word_seq,axis=0)
        pad = torch.from_numpy(padding_(word_seq,500))
        inputs = pad.to(device)
        batch_size = 1
        h = model.init_hidden(batch_size)
        h = tuple([each.data for each in h])
        output, h = model(inputs, h)
        return(output.item())

```

```

[24]: index = 30
print(df['review'][index])
print('='*70)
print(f'Actual sentiment is : {df["sentiment"][index]}')
print('='*70)
pro = predict_text(df['review'][index])
status = "positive" if pro > 0.5 else "negative"
pro = (1 - pro) if status == "negative" else pro

```

```
print(f'Predicted sentiment is {status} with a probability of {pro}')
```

Taut and organically gripping, Edward Dmytryk's *Crossfire* is a distinctive suspense thriller, an unlikely "message" movie using the look and devices of the noir cycle. Bivouacked in Washington, DC, a company of soldiers cope with their restlessness by hanging out in bars. Three of them end up at a stranger's apartment where Robert Ryan, drunk and belligerent, beats their host (Sam Levene) to death because he happens to be Jewish. Police detective Robert Young investigates with the help of Robert Mitchum, who's assigned to Ryan's outfit. Suspicion falls on the second of the three (George Cooper), who has vanished. Ryan slays the third buddy (Steve Brodie) to insure his silence before Young closes in. Abetted by a superior script by John Paxton, Dmytryk draws precise performances from his three starring Bobs. Ryan, naturally, does his prototypical Angry White Male (and to the hilt), while Mitchum underplays with his characteristic alert nonchalance (his role, however, is not central); Young may never have been better. Gloria Grahame gives her first fully-fledged rendition of the smart-mouthed, vulnerable tramp, and, as a sad sack who's leeches into her life, Paul Kelly haunts us in a small, peripheral role that he makes memorable. The politically engaged Dmytryk perhaps inevitably succumbs to sermonizing, but it's pretty much confined to Young's reminiscence of how his Irish grandfather died at the hands of bigots a century earlier (thus, incidentally, stretching chronology to the limit). At least there's no attempt to render an explanation, however glib, of why Ryan hates Jews (and hillbillies and...). Curiously, *Crossfire* survives even the major change wrought upon it -- the novel it's based on (Richard Brooks' *The Brick Foxhole*) dealt with a gay-bashing murder. But homosexuality in 1947 was still Beyond The Pale. News of the Holocaust had, however, begun to emerge from the ashes of Europe, so Hollywood felt emboldened to register its protest against anti-Semitism (the studios always quaked at the prospect of offending any potential ticket buyer). But while the change from homophobia to anti-Semitism works in general, the specifics don't fit so smoothly. The victim's chatting up a lonesome, drunk young soldier then inviting him back home looks odd, even though (or especially since) there's a girlfriend in tow. It raises the question whether this scenario was retained inadvertently or left in as a discreet tip-off to the original engine generating Ryan's murderous rage.

```
=====
Actual sentiment is : positive
```

```
=====
Predicted sentiment is positive with a probability of 0.811938464641571
```

```
[25]: index = 32
print(df['review'][index])
print('='*70)
print(f'Actual sentiment is : {df["sentiment"][index]}')
print('='*70)
pro = predict_text(df['review'][index])
status = "positive" if pro > 0.5 else "negative"
```

```
pro = (1 - pro) if status == "negative" else pro
print(f'predicted sentiment is {status} with a probability of {pro}')
```

My first exposure to the Templarios & not a good one. I was excited to find this title among the offerings from Anchor Bay Video, which has brought us other cult classics such as "Spider Baby". The print quality is excellent, but this alone can't hide the fact that the film is deadly dull. There's a thrilling opening sequence in which the villagers exact a terrible revenge on the Templars (& set the whole thing in motion), but everything else in the movie is slow, ponderous &, ultimately, unfulfilling. Adding insult to injury: the movie was dubbed, not subtitled, as promised on the video jacket.

```
=====
Actual sentiment is : negative
=====
```

```
predicted sentiment is negative with a probability of 0.6419801712036133
```

Some improvement suggestions are as follow:

- Running a hyperparameter search to optimize your configurations.
- Using pretrained word embeddings like Glove word embeddings
- Increasing the model complexity like adding more layers/ using bidirectional LSTMs

[25]: