SENTIMENT CLASSIFICATION ON IMDB MOVIE REVIEWS

Priyanka Gopi, Dept. of Computer Science,

UCFID: 5481740

In this report, we perform sentiment classification using the Large Movie Review Dataset, containing a total of 50000 reviews, comprising of 25000 for training and the other 25000 for testing purpose.

The provided data was present in the form of folders naming train and test within which was present 25000 individual reviews in text files, labelled as positive and negative.

```
import os
import csv
```

```
data_dir = "C:/Users/HP/Downloads/data" #the folder contains the input folders named test and train

def read_imdb_split(split_dir):
    split_dir = os.path.join(data_dir, split_dir)
    texts = []
    labels = []
    for label_dir in ["pos", "neg"]:
        for text_file in os.listdir(os.path.join(split_dir, label_dir)):
            with open(os.path.join(split_dir, label_dir, text_file), "r", encoding="utf-8") as f:
                texts.append(f.read())
                labels.append(0 if label_dir == "neg" else 1)
    return texts, labels
```

The *read_imdb_split()* function, which accepts a directory path as input and reads the text and label data from the files in the subdirectories "pos" and "neg," is defined in the first section of the script. The function outputs two lists: texts and labels, which contain the corresponding binary labels and the raw text data, respectively (0 for negative and 1 for positive).

```
train_texts, train_labels = read_imdb_split("train")
test_texts, test_labels = read_imdb_split("test")
```

```
with open("C:/Users/HP/Downloads/data/imdb_train.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerow(["text", "label"])
    for text, label in zip(train_texts, train_labels):
        writer.writerow([text, label])

with open("C:/Users/HP/Downloads/data/imdb_test.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerow(["text", "label"])
    for text, label in zip(test_texts, test_labels):
        writer.writerow([text, label])
```

Overall, the code reads from the IMDb data, which is divided into subfolders based on the class labels (pos and neg) and saves the text data to CSV files, train.csv and test.csv for further processing.

Following this, the two files were read in another notebook in Colab, to train and test data frames.

```
# Read train data
train = pd.read_csv("/content/drive/MyDrive/data/imdb_train.csv")

# Read test data
test = pd.read_csv("/content/drive/MyDrive/data/imdb_test.csv")
```

```
print("IMDb reviews (combined): train = {}, test = {}".format(train.shape[0], test.shape[0]))
```

```
IMDb reviews (combined): train = 25000, test = 25000
```

Next, we know that the text data typically contains many elements that are useless for model training and is very ill-structured. So, we performed data preprocessing on the review data. Characters like html symbols, emoji representations, numbers, punctuations and stopwords were removed using the function '*clean_input_data'* as provided in below image. This is to make sure that the many stopwords are removed as they carry no meaning in the corpus and the symbols which pretty much don't help while modelling. Stopwords are often used words like "the," "is," "a," etc. that add nothing to the text's meaning. To prevent the model from recognizing any order or pattern in the data, the training and testing sets of data are randomly mixed.

```
def clean_input_data(txt):
        TAG_RE = re.compile(r'<[^>]+>')                                  # html codes
        txt = TAG_RE.sub('', txt.lower())

        txt=txt.encode("ascii","ignore")                                 # emojis
        txt=txt.decode()

        txt=''.join(i for i in txt if not i.isdigit())                   # numbers
        txt = re.sub(r'[^\w\s]', ' ', txt)                               # punctuations

        txt = ' '.join([i for i in txt.split() if not i in STOPWORDS])   # stopwords

        txt=' '.join([i for i in txt.split() if len(i)>2])

        txt=contractions.fix(txt)

        txt=lemmatizer.lemmatize(txt)                                    # lematization
        return txt
```

The data was also shuffled as the labels were in an order where all the positive labelled reviews were on the top of data frame and remaining negative labelled reviews at the end of data frame. For the model to not learn any sort of pattern from the data, the train and test data were shuffled, as shown in the code here.

This ensures the randomness in data.

```
# shuffle the train and test data
from sklearn.utils import shuffle

shuffle(train)
shuffle(test)
```

	text	label
286	first movie ever saw life back years old time ...	1
21974	tagline lucky ones died watching never watched...	0
4144	first entire script mostly improv adding fanta...	1
20883	saw cinema initial release ask world gone mad ...	0
4221	watched gundam time much better gundam wing wa...	1
...	...	...
1407	pleasure seeing saltimbanco live seeing video ...	1

The train and test data were then split in a form to prepare for modelling.

```
#prepare the data for modelling

x_train = train.text
y_train = train.label
x_test = test.text
y_test = test.label
```

The transformers library was utilized to load the *DistilBERT* tokenizer from the pre-trained *"distilbert-base-uncased"* model. The save pretrained method is then used to save the tokenizer to the current directory. The vocab.txt file produced in the preceding step is then used to create a BertWordPieceTokenizer. With the lowercase=True argument, this tokenizer is also configured to lowercase all input text.

```
from tokenizers import BertWordPieceTokenizer
tokenizer = transformers.DistilBertTokenizer.from_pretrained('distilbert-base-uncased' , lower = True)

tokenizer.save_pretrained('.')

fast_tokenizer = BertWordPieceTokenizer('vocab.txt', lowercase=True)
fast_tokenizer
```

```
Downloading (…)solve/main/vocab.txt: 100% 232k/232k [00:00<00:00, 1.77MB/s]
Downloading (…)okenizer_config.json: 100% 28.0/28.0 [00:00<00:00, 1.95kB/s]
Downloading (…)lve/main/config.json: 100% 483/483 [00:00<00:00, 36.0kB/s]
Tokenizer(vocabulary_size=30522, model=BertWordPiece, unk_token=[UNK], sep_token=[SEP], cls_token=[CLS], pad_token=[PAD],
lowercase=True, wordpieces_prefix=##)
```

2. Model Fine-tuning

The corpus data was visualised. Below are the images displaying the frequency of words occurring in positive and negative labelled reviews.

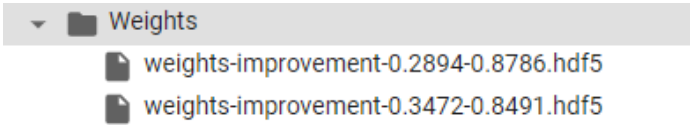
A pre-trained transformer model and a classification layer are added to a TensorFlow BERT model by the build model function. In this example, the transformer model is the TFDistilBertModel from the collection of transformers. Using a pre-trained DistilBERT transformer and a Dense layer with sigmoid activation, this code creates a Keras model. The result is a binary classification prediction with the input shape defined as maxlen.

```
def build_model(transformer, max_len=400):

    input_word_ids = Input(shape=(max_len,), dtype=tf.int32, name="input_word_ids")
    sequence_output = transformer(input_word_ids)[0]
    cls_token = sequence_output[:, 0, :]
    out = Dense(1, activation='sigmoid')(cls_token)

    model = Model(inputs=input_word_ids, outputs=out)
    model.compile(Adam(lr=2e-5), loss='binary_crossentropy', metrics=['accuracy'])

    return model
```

The model is then fit onto the train data as below for 3 epochs, by monitoring the validation loss and validation accuracy. For every imporvement in both the loss and accuracy, the model is saved in the drive path.

```
history = model.fit(x_train,y_train,batch_size = 32 ,validation_data=(x_test,y_test), epochs = 3, callbacks=callbacks_list)
```

```
Epoch 1/3
782/782 [==============================] - ETA: 0s - loss: 0.4127 - accuracy: 0.7988
Epoch 00001: val_loss improved from inf to 0.34720, saving model to /content/drive/MyDrive/Weights/weights-improvement-0.3472-0.8491.hdf5
782/782 [==============================] - 172s 211ms/step - loss: 0.4127 - accuracy: 0.7988 - val_loss: 0.3472 - val_accuracy: 0.8491
Epoch 2/3
782/782 [==============================] - ETA: 0s - loss: 0.2684 - accuracy: 0.8887
Epoch 00002: val_loss improved from 0.34720 to 0.28939, saving model to /content/drive/MyDrive/Weights/weights-improvement-0.2894-0.8786.hdf5
782/782 [==============================] - 164s 209ms/step - loss: 0.2684 - accuracy: 0.8887 - val_loss: 0.2894 - val_accuracy: 0.8786
Epoch 3/3
782/782 [==============================] - ETA: 0s - loss: 0.1916 - accuracy: 0.9258
Epoch 00003: val_loss did not improve from 0.28939
782/782 [==============================] - 161s 206ms/step - loss: 0.1916 - accuracy: 0.9258 - val_loss: 0.3488 - val_accuracy: 0.8586
```



The model was evaluated on the test data and the classification report displays the accuracy of the model's predictions on the test set together with the precision, recall, and F1-score for each class (0 and 1). The model's overall accuracy was 0.84 and its macro-average F1-score was 0.84, indicating that its performance was evenly distributed between the two classes. The model performed similarly across both classes, despite class imbalances.

	precision	recall	f1-score	support
0	0.78	0.96	0.86	12500
1	0.94	0.73	0.82	12500
accuracy			0.84	25000
macro avg	0.86	0.84	0.84	25000
weighted avg	0.86	0.84	0.84	25000

3. Linear probing technique

The model saved earlier was loaded to the model built in new colab notebook. The weights are loaded into the model, which is input from a checkpoint that has already been trained, in previous steps.

```
# load the network weights

filename = "/content/drive/MyDrive/Weights/weights-improvement-0.2894-0.8786.hdf5"
model.load_weights(filename, True)
```

Adam optimizer and binary cross-entropy loss function are used in the model's build. Layers in TensorFlow Keras models cannot use the requires grad attribute because TensorFlow's implementation does not use a computation graph as PyTorch does. When utilizing Keras models with TensorFlow, we can freeze them by setting the trainable attribute of layers to False, preventing weight updates during training. The below output shows how the classifier layer is only not frozen and the remaining are frozen. Linear probing method allows the model to learn task-specific characteristics without forgetting the knowledge acquired from pre-training.

```
[20] # Linear probing - Freeze all layers except the last one
for layer in model.layers[:-1]:
    layer.trainable = False
```

```
# Loop through all the layers in the model
for layer in model.layers:
    # Check if the layer is frozen
    if not layer.trainable:
        print(f"Layer {layer.name} is frozen")
    else:
        print(f"Layer {layer.name} is not frozen")
```

```
Layer input_word_ids is frozen
Layer tf_distil_bert_model is frozen
Layer tf.__operators__.getitem is frozen
Layer dense is not frozen
```

The test set the model's efficiency is assessed, and the accuracy and loss are returned.

```
[22] # Evaluate the performance of the model on the test set
loss, accuracy = model.evaluate(x_test, y_test)
print("Test set loss: {:.4f}, accuracy: {:.4f}".format(loss, accuracy))
```

```
782/782 [==============================] - 362s 459ms/step - loss: 0.2894 - accuracy: 0.8786
Test set loss: 0.2894, accuracy: 0.8786
```

Also, the model's predicted values and the test set's actual labels are used to generate the classification report. As represented below, a test accuracy of 88% was achieved by the linear probed model, which is comparatively better than the baseline and fine-tuned models from previous discussion.

```
[23] from sklearn.metrics import classification_report

y_pred_linearProbing = model.predict(x_test)
y_pred_linearProbing = (y_pred_linearProbing > 0.6)
print(classification_report(y_test, y_pred_linearProbing))
```

```
              precision    recall  f1-score   support

           0       0.86      0.91      0.88     12500
           1       0.90      0.85      0.87     12500

    accuracy                           0.88     25000
   macro avg       0.88      0.88      0.88     25000
weighted avg       0.88      0.88      0.88     25000
```

4. (a) Baseline Model Implementation

Several libraries and packages are imported at the start of the code. To perform natural language processing (NLP) tasks and train the deep learning model in upcoming steps, these are necessary.

```
import numpy as np
import re
import nltk
import pandas as pd
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords
STOPWORDS = set(stopwords.words('english'))
nltk.download('wordnet')
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import contractions
from keras.callbacks import ReduceLROnPlateau
from keras.preprocessing import text, sequence
from keras.models import Sequential
from keras.layers import Embedding, Bidirectional, LSTM, Dense, Dropout
from keras.optimizers import Adam

lemmatizer = WordNetLemmatizer()
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

Next, the pandas library is used to read the training and testing data from their respective CSV files which were obtained from previous step. The testing data simply consists of text reviews, whereas the training data also includes the labels that go along with each review's sentiment/label (positive or negative).

Next, we prepare the data for modelling, wherein, the text data is transformed into numerical form so that the deep learning model may use it. For this, first we transform review data into numerical sequences using the Tokenizer function from the Keras toolkit. The sequences are padded or truncated to a set length using the pad sequences function before being given into the model. The weights of the Embedding layer in the model are initialized using the GloVe word embeddings.

```
# Set the path to the GloVe embeddings file
GLOVE_EMB_DIR = '/content/drive/MyDrive/data/glove.6B.50d.txt'
```

```
[10] def get_coeffs(word, *arr):
         return word, np.asarray(arr, dtype='float32')
     embeddings_dict = dict(get_coeffs(*o.rstrip().rsplit(' ')) for o in open(GLOVE_EMB_DIR))
```

```
[11] all_embs = np.stack(embeddings_dict.values())
     emb_mean, emb_std = all_embs.mean(), all_embs.std()
     embed_size = all_embs.shape[1]

     word_index = tokenizer.word_index
     num_words = min(max_features, len(word_index))

     embedding_matrix = np.random.normal(emb_mean, emb_std, (num_words, embed_size))

     for word, i in word_index.items():

         if i >= num_words: continue
         embedding_vector = embeddings_dict.get(word)
         if embedding_vector is not None: embedding_matrix[i] = embedding_vector
```

```
/usr/local/lib/python3.9/dist-packages/IPython/core/interactiveshell.py:3473: FutureWarning: array
  if (await self.run_code(code, result,  async_=asy)):
```

In the next step, we use the Sequential API from the Keras library to create the deep learning model's architecture. Here, each word in the input sequences is mapped by the model's embedding layer to its matching embedding vector in the embedding matrix. The sequential nature of the input sequences is captured by a bidirectional LSTM layer. To avoid overfitting, a dropout layer is utilized. Predictions are output using two dense layers with ReLU and sigmoid activations. Next, the model is assembled using the accuracy metric, binary cross-entropy loss function, and Adam optimizer as per the requirement.

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 128, 50)           500000

 bidirectional (Bidirectiona  (None, 256)              183296
 l)

 dropout (Dropout)           (None, 256)               0

 dense (Dense)               (None, 16)                4112

 dense_1 (Dense)             (None, 1)                 17

=================================================================
Total params: 687,425
Trainable params: 187,425
Non-trainable params: 500,000
_________________________________________________________________
```

The loss and accuracy statistics for the training and validation sets were tracked after the model had been trained for 10 iterations. The validation set had an accuracy of 85.1% and a loss of 0.3347, whereas the training set had a loss of 0.3289 and an accuracy of 86.1%. The model appears to be functioning well on both the training and validation sets, according to these data, and it is not overfitting.

```
batch_size = 256
epochs=10
embed_size=50

history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, validation_data=(X_test, y_test), callbacks=[learning_rate_reduction])
```

```
Epoch 1/10
98/98 [==============================] - 209s 2s/step - loss: 0.6091 - accuracy: 0.6656 - val_loss: 0.5056 - val_accuracy: 0.7606 - lr: 0.0020
Epoch 2/10
98/98 [==============================] - 185s 2s/step - loss: 0.4913 - accuracy: 0.7692 - val_loss: 0.4396 - val_accuracy: 0.7938 - lr: 0.0020
Epoch 3/10
98/98 [==============================] - 222s 2s/step - loss: 0.4355 - accuracy: 0.8065 - val_loss: 0.3947 - val_accuracy: 0.8231 - lr: 0.0020
Epoch 4/10
98/98 [==============================] - 170s 2s/step - loss: 0.4030 - accuracy: 0.8230 - val_loss: 0.3836 - val_accuracy: 0.8333 - lr: 0.0020
Epoch 5/10
98/98 [==============================] - 202s 2s/step - loss: 0.3887 - accuracy: 0.8300 - val_loss: 0.3800 - val_accuracy: 0.8298 - lr: 0.0020
Epoch 6/10
98/98 [==============================] - 203s 2s/step - loss: 0.3749 - accuracy: 0.8398 - val_loss: 0.4075 - val_accuracy: 0.8233 - lr: 0.0020
Epoch 7/10
98/98 [==============================] - 203s 2s/step - loss: 0.3587 - accuracy: 0.8477 - val_loss: 0.3517 - val_accuracy: 0.8460 - lr: 0.0020
Epoch 8/10
98/98 [==============================] - 203s 2s/step - loss: 0.3553 - accuracy: 0.8492 - val_loss: 0.3481 - val_accuracy: 0.8507 - lr: 0.0020
Epoch 9/10
98/98 [==============================] - 173s 2s/step - loss: 0.3355 - accuracy: 0.8608 - val_loss: 0.3398 - val_accuracy: 0.8528 - lr: 0.0020
Epoch 10/10
98/98 [==============================] - 203s 2s/step - loss: 0.3289 - accuracy: 0.8618 - val_loss: 0.3347 - val_accuracy: 0.8519 - lr: 0.0020
```

The classification report function from the sklearn.metrics library was used to further assess the performance of the model. For each class in the classification job, this function offers a full report on the precision, recall, F1-score, and support. The function took the arrays y_test and y_pred as inputs.

```
from sklearn.metrics import classification_report

y_pred = model.predict(X_test)
y_pred = (y_pred > 0.6)
print(classification_report(y_test,y_pred))
```

```
782/782 [==============================] - 91s 116ms/step
              precision    recall  f1-score   support

           0       0.81      0.90      0.85     12500
           1       0.89      0.78      0.83     12500

    accuracy                           0.84     25000
   macro avg       0.85      0.84      0.84     25000
weighted avg       0.85      0.84      0.84     25000
```

A test accuracy of 84% was achieved. Overall, these findings imply that the model is well accurate and precise at classifying movie reviews as favourable or negative. There is scope for additional testing and tuning for better model resilience.

By adding a single dense layer and training the entire model on the task-specific dataset, the provided code adjusts the frozen layers of a pre-trained DistilBERT model for sentiment analysis. The fine-tuning technique, as compared to the baseline model, gave the model the opportunity to learn from the IMDb dataset, which led to an increase in accuracy between the **baseline** and **fine-tuned** models, with accuracy, 84% and loss of 0.33 and 85% accuracy with a loss of 0.2, respectively. The accuracy of the **linear probed** model yielded a value of 87.86% as opposed to 84% without linear probing. With this model, we could use the model's previously acquired information while also training just the classifier layer on the task-specific data. This method achieves a compromise between using previously learned information and capturing task-specific properties, improving performance on the downstream job. This increase in accuracy shows that the linear probing strategy works well for optimizing the pre-trained layers and enhancing the model's performance on a particular task.

Thus, compared to the baseline model, the linear probed worked much better and fine-tuned model almost gave similar results.