

Part 1: Train a neural network for sentiment analysis

```
from google.colab import drive

drive.mount('/content/gdrive')

import pandas as pd

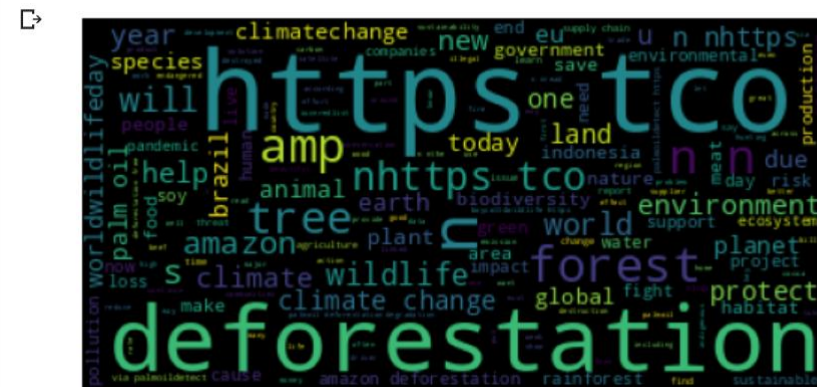
traindata = pd.read_csv('gdrive/My Drive/deforestation_sentiment_train.csv', usecols=["text", "distillbert_valence"], na_values = ['no info', '.'])
testdata = pd.read_csv('gdrive/My Drive/deforestation_sentiment_val.csv', usecols=["text", "distillbert_valence"], na_values = ['no info', '.'])
```

Mounted at /content/gdrive

The two csv files were saved on my gdrive so I created code to get the data from my gdrive and read and save it to the variables `traindata` and `testdata`. Then I selected `text` as an input and `distillbert_valence` as an output, as well as assigned data for null cells.

```
text = traindata['text'].values

import matplotlib.pyplot as plt
from wordcloud import WordCloud, STOPWORDS
wordcloud = WordCloud().generate(str(text))
plt.imshow(wordcloud)
plt.axis("off")
plt.show()
```



I created a variable called `text` and stored the data from the `text` column from the `csv` file. Then used `matplotlib` to create a word cloud of the positive and negative tweets posted by twitter users. This is an optional analysis step that gives a feel for the word frequencies in the data.

```
import re

def process_sentence(sentence):
    return re.sub(r'[\\\/:*«`\'?¿";!<>,.|]', '', sentence.lower().strip())
```

I imported the regular expressions package and created a method to preprocess the dataset to make tokens as uniform as possible. This is by removing special characters, like punctuation marks, white spaces and making all words lower case.

```

▶ def convert_decimals(decimals):
    converted_decimals = []
    for decimal in decimals:
        if decimal < 0:
            converted_decimals.append("negative")
        else:
            converted_decimals.append("positive")

    return converted_decimals

```

I created a method that changes floating point numbers to the strings negative and positive depending on whether they are positive or negative numbers.

```

▶ decimals = traindata["distillbert_valence"]
converted_decimals_train = convert_decimals(decimals)

decimals = testdata["distillbert_valence"]
converted_decimals_test = convert_decimals(decimals)

```

I applied the method to both the traindata and testdata distillbert_valence columns

```

[37] X_train = traindata["text"].apply(process_sentence)
      Y_train = converted_decimals_train

      X_test = testdata["text"].apply(process_sentence)
      Y_test = converted_decimals_test

```

I applied my method to the text column in both pandas data frames to make all tweets lower case and assigned it to X for training. I then assigned the distillbert_valence column to Y to use as the labels.

```

▶ labels = set(Y_train)

```

I specify that the labels that will be used correspond to the set of individual strings in y train data.

```

[11] elements = (' '.join([sentence for sentence in X_train])).split()
      elements.append("<UNK>")

```

I split the training text data into strings that correspond to individual words to further analyze the tweets and added a backup character to use in case it comes across anything unseen.

```
[12] def create_lookup_tables(text):
      vocab = set(text)
      vocab_to_int = {word: i for i, word in enumerate(vocab)}
      int_to_vocab = {v:k for k, v in vocab_to_int.items()}
      return vocab_to_int, int_to_vocab
```

I created a method for a lookup table and specified that the variable vocab corresponds to the set of individual strings in the input text. The lookup tables allow for the words to be converted to integer then back to words, making it easy for the learning model to understand.

```
[13] vocab_to_int, int_to_vocab = create_lookup_tables(elements)
      labels_to_int, int_to_labels = create_lookup_tables(Y_train)
```

I applied the lookup table method to both elements and Y to convert between numbers and words from the train dataset.

```
print("Vocabulary of our dataset: {}".format(len(vocab_to_int)))
```

Vocabulary of our dataset: 8422

I printed the size of the vocabulary for the element's dataset

```
def convert_to_int(data, data_int):
    all_items = []
    for sentence in data:
        all_items.append([data_int[word] if word in data_int else data_int["<UNK>"] for word in sentence.split()])
    return all_items
```

I created a function that uses the lookup tables to convert each tweet text into a numeric representation to one hot encode the data

```
[16] X_test_encoded = convert_to_int(X_test, vocab_to_int)
      X_train_encoded = convert_to_int(X_train, vocab_to_int)

      Y_data = convert_to_int(Y_train, labels_to_int)
```

I applied the function to the x data for train and text and the y train data.

```
[17] from sklearn.preprocessing import OneHotEncoder
      enc = OneHotEncoder()
      enc.fit(Y_data)
```

I imported sklearn's onehotencoder and fitted it on y data so it can see what categories there are and what shape the resulting matrix needs to take.

```
[18] Y_train_encoded = enc.fit_transform(convert_to_int(Y_train,
labels_to_int)).toarray()
Y_test_encoded = enc.fit_transform(convert_to_int(Y_test,
labels_to_int)).toarray()
```

I applied the encoder to y train and test data and converted the resulting representations into arrays, as this is the format keras will need

```
print(Y_train_encoded[:10], '\n', Y_train[:10])
```

```
[[0. 1.]
 [0. 1.]
 [0. 1.]
 [0. 1.]
 [0. 1.]
 [0. 1.]
 [0. 1.]
 [1. 0.]
 [0. 1.]
 [1. 0.]]
['negative', 'negative', 'negative', 'negative', 'negative', 'negative', 'negative', 'positive', 'negative', 'positive']
```

I displayed a sample of the encoded data and normal data to check if it has been correctly encoded

```
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import SimpleRNN, LSTM, GRU, Embedding
from keras.preprocessing import sequence

max_sentence_length = 200
embedding_vector_length = 300
dropout = 0.5
```

I imported tensor flow and other packages and specified hyperparameters. The sequence length has been set to 200 and captures an average amount of linguistic content and does not take too long to train. The embedding vector also captures an average context of co-occurring words, and the 0.5 drop out helps prevent the model from overfitting.

```
[22] X_train_pad = keras.preprocessing.sequence.pad_sequences(X_train_encoded, maxlen=max_sentence_length)
X_test_pad = keras.preprocessing.sequence.pad_sequences(X_test_encoded, maxlen=max_sentence_length)
```

I padded the x train and test data to the length of the max sentence length as keras cannot deal with variably sized sequences. This pads up sequences with random symbols that are shorter than the set max sentence length parameter.

```
model = Sequential()

model.add(Embedding(len(vocab_to_int), embedding_vector_length, input_length=max_sentence_length))
model.add(LSTM(256, return_sequences=True, dropout=dropout, recurrent_dropout=dropout))
model.add(LSTM(256, dropout=dropout, recurrent_dropout=dropout))
model.add(Dense(len(labels), activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

print(model.summary())

model.fit(X_train_pad, Y_train_encoded, epochs=10, batch_size=256, validation_data=(X_test_pad, Y_test_encoded))

scores = model.evaluate(X_test_pad, Y_test_encoded, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

I changed the hyperparameters multiple times but the best result I was able to get was with a max sentence length of 200, an embedding vector length of 300 and a dropout of 0.5. I initialized the model and added an embedding layer that maps the length of the vocabulary to the embedding vector length. I also added two long short-term memory layers and a dense layer and compiled the model. I printed the model summary and trained the model with the train data and used the x test pad and y test pad for the validation. The metric I used was accuracy as the data only has two classes and it helps by determining the percentage of correct predictions made by the model. Lastly, I evaluated the model and displayed its overall accuracy.

Model: "sequential_4"

Layer (type)	Output Shape	Param #
embedding_4 (Embedding)	(None, 200, 300)	2526600
lstm_6 (LSTM)	(None, 200, 256)	570368
lstm_7 (LSTM)	(None, 256)	525312
dense_4 (Dense)	(None, 2)	514

=====
Total params: 3,622,794
Trainable params: 3,622,794
Non-trainable params: 0

None
Epoch 1/10
4/4 [=====] - 61s 13s/step - loss: 0.6411 - accuracy: 0.7093 - val_loss: 0.5516 - val_accuracy: 0.7460
Epoch 2/10
4/4 [=====] - 49s 12s/step - loss: 0.5253 - accuracy: 0.7604 - val_loss: 0.5551 - val_accuracy: 0.7460
Epoch 3/10
4/4 [=====] - 51s 13s/step - loss: 0.5002 - accuracy: 0.7604 - val_loss: 0.5334 - val_accuracy: 0.7460
Epoch 4/10
4/4 [=====] - 50s 12s/step - loss: 0.4392 - accuracy: 0.7785 - val_loss: 0.5241 - val_accuracy: 0.7621
Epoch 5/10
4/4 [=====] - 51s 13s/step - loss: 0.2928 - accuracy: 0.8829 - val_loss: 0.6180 - val_accuracy: 0.8105
Epoch 6/10
4/4 [=====] - 49s 12s/step - loss: 0.1397 - accuracy: 0.9606 - val_loss: 0.6365 - val_accuracy: 0.7661
Epoch 7/10
4/4 [=====] - 52s 13s/step - loss: 0.0708 - accuracy: 0.9787 - val_loss: 0.7276 - val_accuracy: 0.7661
Epoch 8/10
4/4 [=====] - 52s 13s/step - loss: 0.0336 - accuracy: 0.9904 - val_loss: 0.8199 - val_accuracy: 0.7944
Epoch 9/10
4/4 [=====] - 49s 13s/step - loss: 0.0239 - accuracy: 0.9925 - val_loss: 0.8586 - val_accuracy: 0.7863
Epoch 10/10
4/4 [=====] - 52s 13s/step - loss: 0.0151 - accuracy: 0.9968 - val_loss: 0.7641 - val_accuracy: 0.7823
Accuracy: 78.23%

Model: "sequential_3"

Layer (type)	Output Shape	Param #
embedding_3 (Embedding)	(None, 200, 300)	2526600
simple_rnn (SimpleRNN)	(None, 200, 256)	142592
simple_rnn_1 (SimpleRNN)	(None, 256)	131328
dense_3 (Dense)	(None, 2)	514

=====
Total params: 2,801,034
Trainable params: 2,801,034
Non-trainable params: 0

None
Epoch 1/10
4/4 [=====] - 15s 3s/step - loss: 0.8740 - accuracy: 0.5048 - val_loss: 0.7497 - val_accuracy: 0.3952
Epoch 2/10
4/4 [=====] - 12s 3s/step - loss: 0.8411 - accuracy: 0.5037 - val_loss: 0.6922 - val_accuracy: 0.5242
Epoch 3/10
4/4 [=====] - 10s 3s/step - loss: 0.8055 - accuracy: 0.5293 - val_loss: 0.6539 - val_accuracy: 0.6411
Epoch 4/10
4/4 [=====] - 12s 2s/step - loss: 0.7610 - accuracy: 0.5293 - val_loss: 0.6049 - val_accuracy: 0.7500
Epoch 5/10
4/4 [=====] - 12s 3s/step - loss: 0.7067 - accuracy: 0.5921 - val_loss: 0.5778 - val_accuracy: 0.7460
Epoch 6/10
4/4 [=====] - 12s 3s/step - loss: 0.7129 - accuracy: 0.5793 - val_loss: 0.5715 - val_accuracy: 0.7460
Epoch 7/10
4/4 [=====] - 13s 3s/step - loss: 0.7075 - accuracy: 0.5953 - val_loss: 0.5720 - val_accuracy: 0.7460
Epoch 8/10
4/4 [=====] - 14s 3s/step - loss: 0.6580 - accuracy: 0.6262 - val_loss: 0.5855 - val_accuracy: 0.7460
Epoch 9/10
4/4 [=====] - 12s 3s/step - loss: 0.6445 - accuracy: 0.6571 - val_loss: 0.6101 - val_accuracy: 0.7460
Epoch 10/10
4/4 [=====] - 11s 3s/step - loss: 0.6467 - accuracy: 0.6411 - val_loss: 0.6486 - val_accuracy: 0.7460
Accuracy: 74.60%

I decided to use a long short-term memory network because it is typically used for text processing. It is

also better when compared to using a simple recurrent neural network as it uses special units in addition to standard units and is efficient at modelling complex sequential data (Great Learning Team, 2022). They have a memory cell that remembers data for extended periods of time and therefore are much better at handling long-term dependencies (Great Learning Team, 2022). They may be slow compared to using a simple recurrent neural network but from testing both out, long short-term memory network had a higher accuracy than simple neural network. LSTM had an accuracy of 78.23% and SimpleRNN had an accuracy of 74.60%. The results are above.

Part 2: Vary the knowledge representation

```
▶ #!/wget http://nlp.stanford.edu/data/glove.6B.zip
#!/unzip -q glove.6B.zip
path_to_glove_file = "/content/glove.6B.100d.txt"
```

I downloaded pre-trained GloVe embeddings and unzipped the downloaded file

```
▶ embedding_matrix = np.zeros((num_tokens, embedding_dim))

for word, i in vocab_to_int.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
        hits += 1
    else:
        misses += 1
print("Converted %d words (%d misses)" % (hits, misses))
```

```
📄 Converted 4464 words (3958 misses)
```

I used the embeddings to create an embedding index which matches words in the vocabulary to their respective embedding representation. I then created an embedding matrix from the embedding index, which converts the words it can and reports back matching words and non-matching words.

```

model = Sequential()

embedding_layer = Embedding(num_tokens, embedding_dim, embeddings_initializer=keras.initializers.Constant(embedding_matrix), trainable=True)
model.add(embedding_layer)
model.add(LSTM(256, return_sequences=True, dropout=dropout, recurrent_dropout=dropout))
model.add(LSTM(256, dropout=dropout, recurrent_dropout=dropout))
model.add(Dense(len(labels), activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

print(model.summary())

model.fit(X_train_pad, Y_train_encoded, epochs=10, batch_size=32, validation_data=(X_test_pad, Y_test_encoded))

scores = model.evaluate(X_test_pad, Y_test_encoded, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))

```

I defined a new embedding layer based on the embedding matrix and used the same model from my neural network for training.

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, None, 100)	842400
lstm (LSTM)	(None, None, 256)	365568
lstm_1 (LSTM)	(None, 256)	525312
dense (Dense)	(None, 2)	514

Total params: 1,733,794
 Trainable params: 1,733,794
 Non-trainable params: 0

```

None
Epoch 1/10
30/30 [=====] - 139s 4s/step - loss: 0.5846 - accuracy: 0.7529 - val_loss: 0.5576 - val_accuracy: 0.7460
Epoch 2/10
30/30 [=====] - 129s 4s/step - loss: 0.5094 - accuracy: 0.7678 - val_loss: 0.5028 - val_accuracy: 0.7540
Epoch 3/10
30/30 [=====] - 132s 4s/step - loss: 0.4547 - accuracy: 0.7859 - val_loss: 0.4971 - val_accuracy: 0.7742
Epoch 4/10
30/30 [=====] - 127s 4s/step - loss: 0.4027 - accuracy: 0.8275 - val_loss: 0.5023 - val_accuracy: 0.7540
Epoch 5/10
30/30 [=====] - 132s 4s/step - loss: 0.3243 - accuracy: 0.8797 - val_loss: 0.5349 - val_accuracy: 0.7782
Epoch 6/10
30/30 [=====] - 129s 4s/step - loss: 0.2923 - accuracy: 0.8775 - val_loss: 0.6494 - val_accuracy: 0.7863
Epoch 7/10
30/30 [=====] - 129s 4s/step - loss: 0.2096 - accuracy: 0.9148 - val_loss: 0.4956 - val_accuracy: 0.7702
Epoch 8/10
30/30 [=====] - 128s 4s/step - loss: 0.1824 - accuracy: 0.9329 - val_loss: 0.5625 - val_accuracy: 0.7823
Epoch 9/10
30/30 [=====] - 129s 4s/step - loss: 0.1304 - accuracy: 0.9521 - val_loss: 0.6802 - val_accuracy: 0.7742
Epoch 10/10
30/30 [=====] - 122s 4s/step - loss: 0.1107 - accuracy: 0.9563 - val_loss: 0.6852 - val_accuracy: 0.7944
Accuracy: 79.44%

```

After 10 epochs, I got a validation accuracy of 79.44%. Based on this, the embeddings have helped my model from its baseline of 78.23% and is better to use than a simple embedding layer.

The model I chose for my embedding layer was GloVe. GloVe is used for word representation and stands for global vectors. It is a type of word embedding that encodes the co-occurrence probability of two words as vector differences and uses the Euclidean separation between two words to evaluate the semantic similarity of the corresponding words (Jeffrey P., Richard S., and Christopher D. M., 2014). I decided to use GloVe because even though it is slower than Word2vec, it includes global statistics to obtain word vectors, but Word2vec only relies on local statistics. GloVe also creates a lower-dimensional matrix hence a preferable word embedding can be acquired by reducing the modification loss and is

easier to train over more data (Thushan G., 2019). Lastly, GloVe seems to perform better in word relations and named entity identification problems.

Language models lack understanding, they manipulate tokens but have no semantic anchoring for what they say or do. False or misleading information could be given out when fed into the model as the model does not know what the difference between right and wrong data is and personal data could be leaked. They can create unfair discrimination as there are stereotypes and social biases that can deny or burden identities that differ. For example, a normal family consists of a child, a female mother, and a male father. Deep learning models work as black boxes and supply no reasoning for their decisions although they assist in providing solutions. These models benefit users and help serve the public good but could do harm only if used in the wrong hands. They also leave a large carbon footprint when being used to train data which could be as many as five cars in their lifetime (Karen H., 2019). They also risk harming marginalized communities by reinforcing hegemonic viewpoints. It takes substantial amounts of resources and time to train language models from scratch compared to pre-trained language models. With pre-trained models, the weights and features can be used as a starting point. It is also a low-cost investment model as it needs less data and computational power to train the model and saves time and money.

Part 3: Create a probabilistic baseline

```
✓ [10] vectorizer = CountVectorizer()  
0s      train_features = vectorizer.fit_transform(X_train)  
      test_features = vectorizer.transform(X_test)
```

I used the CountVectorizer from sk-learn to convert the tweet text into a matrix of token counts. I then fitted the vectorizer on the X_train data to transform the tweets into feature vectors. I also transformed the X_test data into feature vectors using the vocabulary learned from the X_train data.

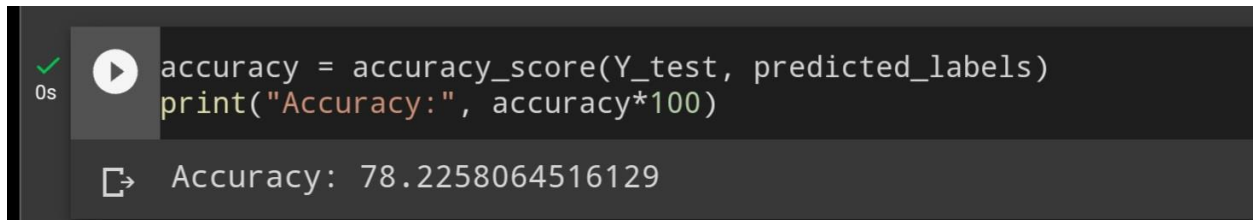
```
✓ [11] classifier = MultinomialNB()  
0s      classifier.fit(train_features, Y_train)
```

```
▸ MultinomialNB  
MultinomialNB()
```

I trained in a MultinomialNB classifier, which is a probabilistic classifier based on the Naive Bayes algorithm, using the train features and Y_train labels. This trains the classifier on the training data, allowing it to learn the relationship between the features and the Y_train labels.

```
✓ [12] predicted_labels = classifier.predict(test_features)  
0s
```

I created a variable that predicts the labels for the test data. This assigns labels to the testing data based on the learned model.

A screenshot of a Jupyter Notebook cell. On the left, there is a green checkmark icon and a play button icon. The code in the cell is:

```
accuracy = accuracy_score(Y_test, predicted_labels)
print("Accuracy:", accuracy*100)
```

 Below the code, the output is displayed:

```
Accuracy: 78.2258064516129
```

Finally, I used the trained classifier to predict the labels for the test features and calculated the accuracy of the classifier by comparing the predicted labels with the actual labels. The accuracy is calculated as the ratio of correctly classified samples to the total number of samples in the test data.

This model had an accuracy of 78.23%, which is the same percentage as using LSTM with a simple embedding layer. It has a lower accuracy compared to GloVe with a 79.44% accuracy. From this analysis, GloVe is the model that has the best accuracy and will give better results.

References

- Great Learning Team. (2022) Types of Neural Networks and Definition of Neural Network. Great Learning. 23 November. Available online: <https://www.mygreatlearning.com/blog/types-of-neural-networks/> [Accessed 20/05/2023]
- Jeffrey P., Richard S., and Christopher D. M. (2014) GloVe: Global Vectors for Word Representation. Available online: <https://nlp.stanford.edu/projects/glove/> [Accessed 02/06/2023]
- Thushan G. (2019) Intuitive Guide to Understanding GloVe Embeddings. Understanding theory behind GloVe and Keras implementation! Available online: <https://towardsdatascience.com/light-on-math-ml-intuitive-guide-to-understanding-glove-embeddings-b13b4f19c010> [Accessed 02/06/2023]
- Karen H. (2019) Training a single AI model can emit as much carbon as five cars in their lifetimes. Deep learning has a terrible carbon footprint. Available online: <https://www.technologyreview.com/2019/06/06/239031/training-a-single-ai-model-can-emit-as-much-carbon-as-five-cars-in-their-lifetimes/> [Accessed 02/06/2023]