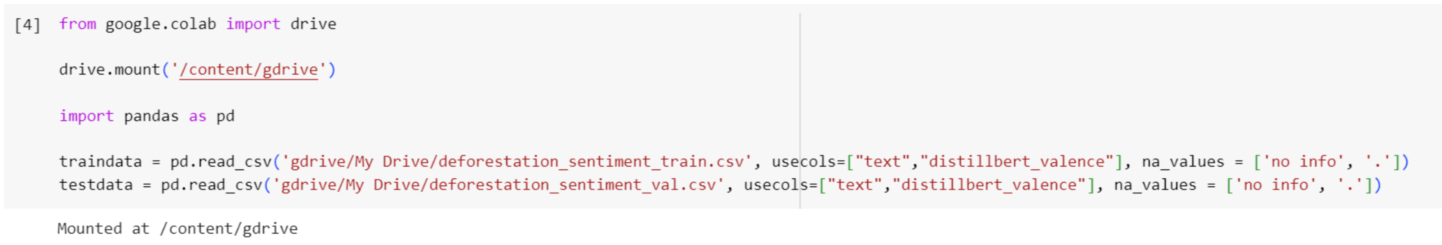
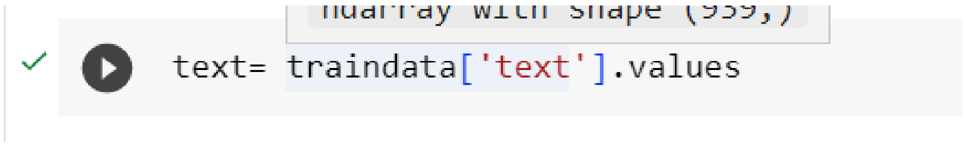
Part 1: Train a neural network for sentiment analysis



```
[4]  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
```

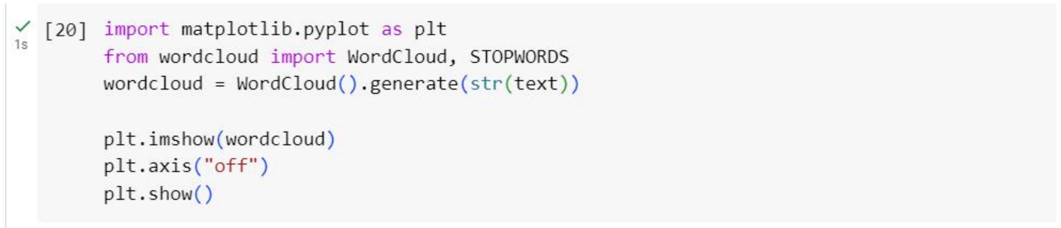
I uploaded the datasets on Gdrive and used the columns “text” (i.e. our data) and “distillbert_valence” (as our labels). Further on, I imported pandas to read the csv files, converted the data into pandas data Frames by using ”pd.read_csv,” and, after csv files being read, they got stored it in”traindata ” and ”testdata“(as dataframes).



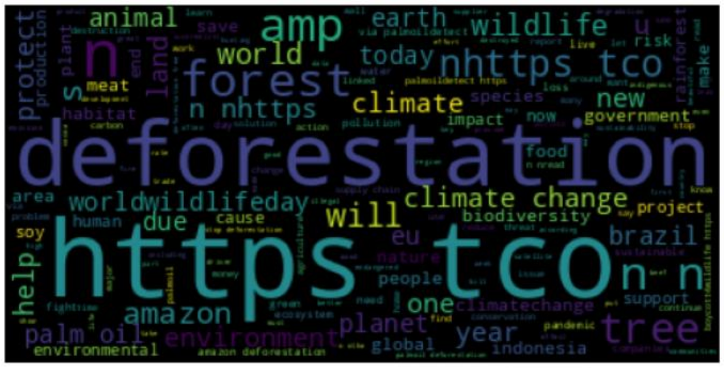
```
ndarray with shape (939,)
text= traindata['text'].values
```

The tweets that are called ‘text’ in the the “traindata” df are stored in a new variable called as text and each tweet is printed by using “ .values”.

<u>DATA VISUALIZATION</u>

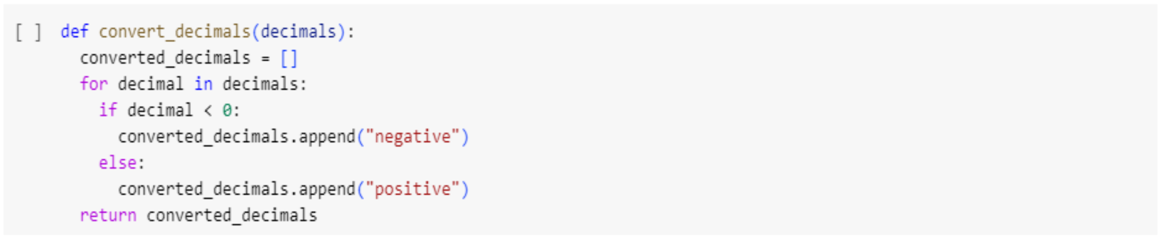


```
[20] import matplotlib.pyplot as plt
     from wordcloud import WordCloud, STOPWORDS
     wordcloud = WordCloud().generate(str(text))

     plt.imshow(wordcloud)
     plt.axis("off")
     plt.show()
```



Later, I wanted to display the frequency of each occurrence of a word in tweets through wordcloud by importing the matplotlib library . By using the “text” variable (as it represents the tweets) and here “.plt” and “plt.show” is used to display it after I imported the respective wordcloud. wordcloud() function will generate a word cloud.

The role of axis off is to turn off the x and y axis that would have appeared in the first place that is irrelevant here so had it turn it off.



```
[ ] 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 function called convert_decimals that takes one parameter decimals and its purpose is to convert a list of decimal numbers into a list of strings based on their sign positive or negative. An empty list called converted_decimals will store the converted values.

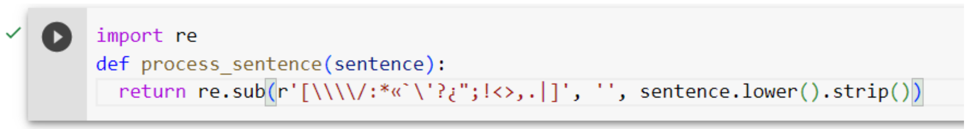
for decimal in decimals - This line starts a for loop to go thru each element decimal number in input.

if decimal < 0 -if it’s negative, the string "negative" gets added to the converted_decimals list using the append() method in “converted_decimals.append("negative")”.

Else - If the condition is not true then it goes to the next else block. In the else block, the code adds the string "positive" to the converted_decimals list using the method “converted_decimals.append("positive") -”.

At Last, function reaches the return statement.return converted_decimals as This line returns the list converted_decimals, which contains the converted values which will either be “positive” or “negative” depending on the sign of the corresponding decimal value in the input list .

DATA PREPROCESSING



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

Now for data preprocessing, I created a method process_sentence() that is ultimately converting the lines (tweet sentence) as an input sentence into clean texts as an output by removing all special characters from the sentence.

The re.sub (\\\\/:*«`\'?¿";!<>,.|) will return an output string by replacing all the special characters in the input sentence with an empty string, effectively removing them from the sentence in return.

“.strip()” removes extra space including any leading and trailing whitespace and “sentence.lower()” making the string lowercase and only returns any word in lowercase in case there was any uppercase word. I am using the re package for more powerful pattern matching capabilities for more complex string.

```
X_train = traindata["text"].apply(process_sentence)
y_train= convert_decimals(traindata["distillbert_valence"])

X_test = testdata["text"].apply(process_sentence)
y_test= convert_decimals(testdata["distillbert_valence"])

labels= set(y_train)
```

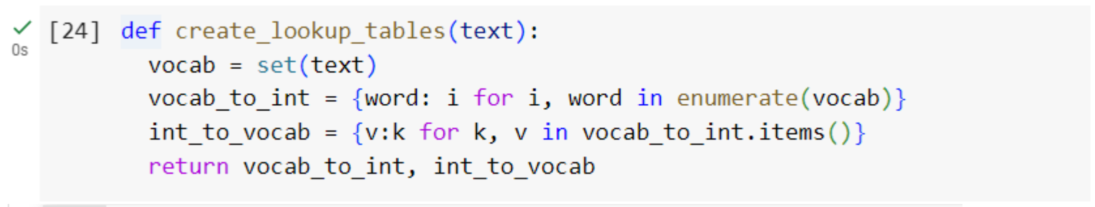
I created a new DataFrame called X_train .I tried to apply the function process_sentence to the "text" column of the DataFrame traindata. The apply() method is applying the process_sentence to each element of the "text" column, and the output is stored in X_train. The process_sentence function cleans the text as explained above.

Another dataframe that I created called ”y_train” is calling the function convert_decimals on the "distillbert_valence" column of the DataFrame traindata. To convert the values in the "distillbert_valence" column to decimals (floating-point numbers) and store them in y_train.

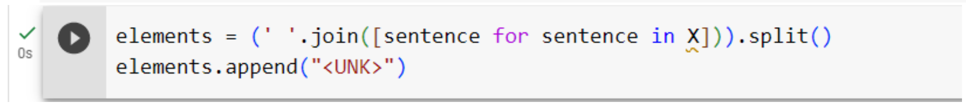
Same thing goes on for X-test and Y_test below.

All the converted positive and negative strings that are in “y_train” are stored in labels.

# Split all our sentences



```
[24] 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
```



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

Here, I am adding the entire tweet sentences (clean processed tweet) by using ”.join” and adding them up by using a whitespace character represented by ” ” and then later on splitting them into words. elements then contain all my X data split into words. Then I used ”<UNK>” wich is appending any unnoticed literal and adding to the vocab.

<u>Training a Recurrent Neural Network:</u>

1. Making a vocabulary...
For vocabulary, we are using the elements from above..

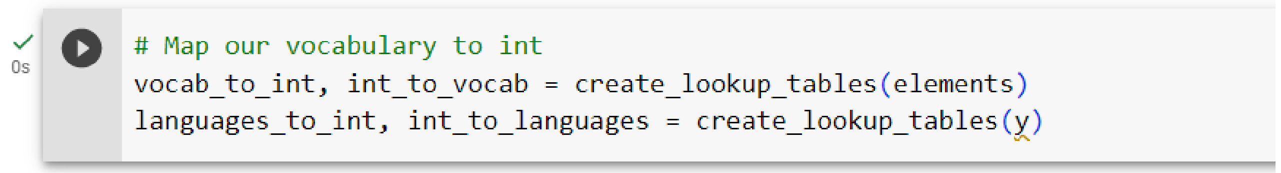
2. Look-up tables

```
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
```

For creating look-up tables, I created a function called create_lookup_tables(text) that takes the text as a parameter and removes all the duplicated string values from the text by using set() and stores them in "vocab" variable.

I then created dictionaries that map a word to an integer in the format of a dictionary like key and value and store it in vocab_to_int variable as it will be useful for machine learning models. The enumerate() function is used to iterate over the words in the vocab set, and it provides an index i for each word.

int_to_vocab is a dictionary below that maps each integer value to its respective word in the vocabulary. It does this by swapping the keys and values of the vocab_to_int dictionary .In the end both lookup tables are created. These convert words to integers and vice versa, which is useful during data preprocessing and model training in NLP tasks.



```
# Map our vocabulary to int
vocab_to_int, int_to_vocab = create_lookup_tables(elements)
languages_to_int, int_to_languages = create_lookup_tables(y)
```

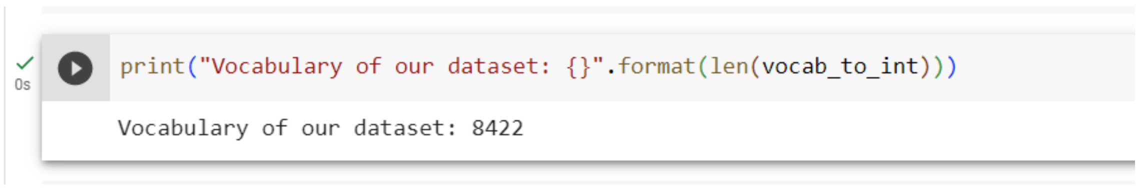
created 4 Lookup tables:

In the first phrase-

Vocab_to_int – creates a lookup table where the tokenized words that are "elements" which get converted into unique integers using dictionaries.

Int_to_vocab – creates a lookup table/dictionary where it maps each integer value back to its corresponding word/token in the vocabulary and reverse of it can also take place.

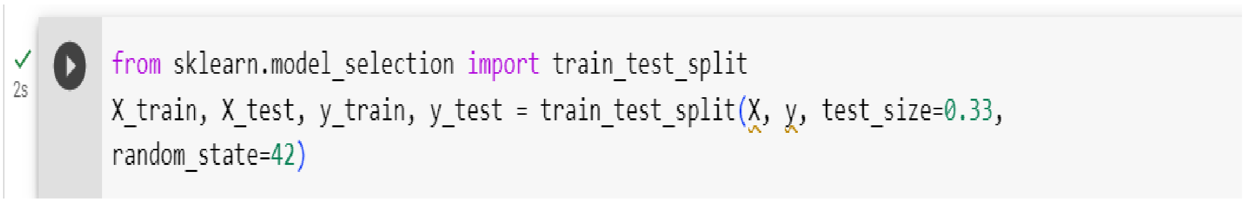
Language_to_int – creates a lookup table where the labels that are parameter "y" get converted to unique integer from y_train. And int_to_languages is just the opposite as explained above.



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

```
Vocabulary of our dataset: 8422
```

So, the size of the vocab of vocab_to_int is 8422.



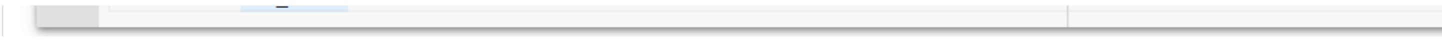
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
random_state=42)
```

I split my data into a train and test set. I believed It's important to keep these two apart. Python's sklearn package does this easily so I used it. X_train will be the training data (i.e. tweets) used during the training phase, while y_train will be the labels (positive, negative) used during training. X_test and y_test is used out at this stage so I can use them for evaluation later on.

3. Represent data numerically...

One-hot encoding...

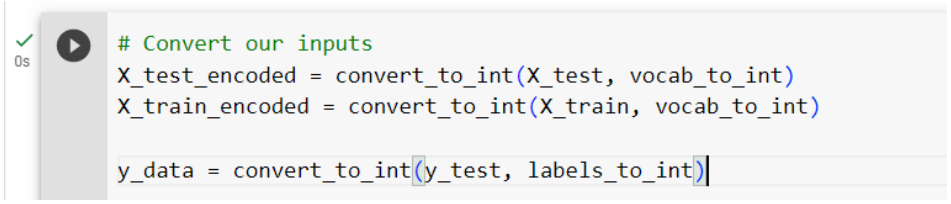
I've performed tasks for an integer representation of our sentences, what Keras needs is a one-hot encoding that is table with 0s and 1s, where a 1 indicates the position in which a word is present



Here it's basically creating an empty list that is checking if the word is present in the data_int lookup table and converting it into its corresponding integer value from my tweets data and storing it in the list "all_items" and if it didn't contain it in then store it in data_ink['<UNK>']

return all_items: Finally, the function returns the **all_items** list, which contains the one-hot encoded representation of the input text data which is also the one hot encoded version of each word in integer format.

Next step is converting our inputs that are X_test_encoded and X_train_encoded by using the above methods we created by integrating it with our vocab :



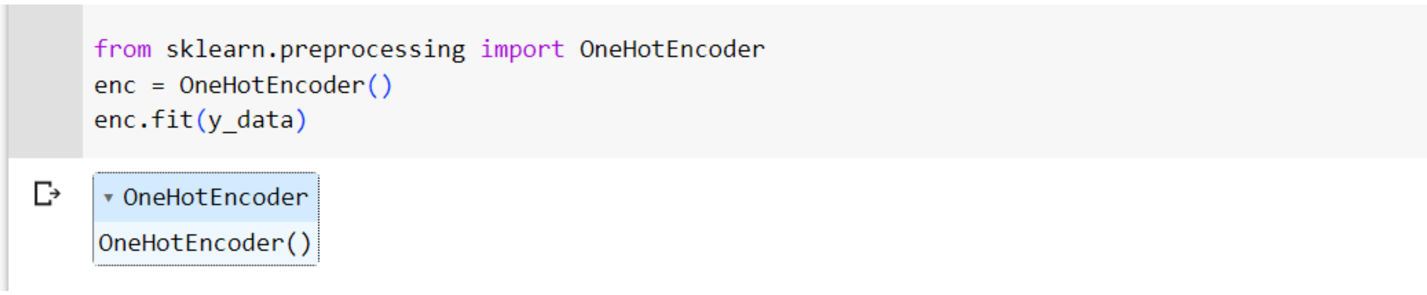
```
# Convert our inputs
X_test_encoded = convert_to_int(X_test, vocab_to_int)
X_train_encoded = convert_to_int(X_train, vocab_to_int)

y_data = convert_to_int(y_test, labels_to_int)
```

I used sklearn's OneHotEncoder to represent categorical data in a one-hot encoding matrix of zeros and ones. As the model will learn on how to change a particular word into the format of 0s and ones and after learning it, it will try to implement it in the upcoming models we trained it to. The encoder is imported and fitted on y_data (so it can see what categories they are, and hence what shape the resulting matrix needs to take).

enc creates an instance of the OneHotEncoder and assigns it to the variable enc.

"enc.fit(y_data)" fits the one-hot encoder to variable y_data. The fit method is used to analyze the data and learn the unique categories present in y_data. This step is crucial to determine the unique labels/classes in y_data, which will be used to create the one-hot encoding scheme.

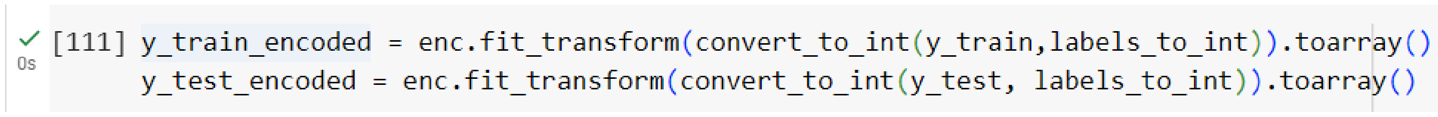
After executing the code, the enc object is ready to transform data using the learned



```
from sklearn.preprocessing import OneHotEncoder
enc = OneHotEncoder()
enc.fit(y_data)
```

```
▾ OneHotEncoder
OneHotEncoder()
```

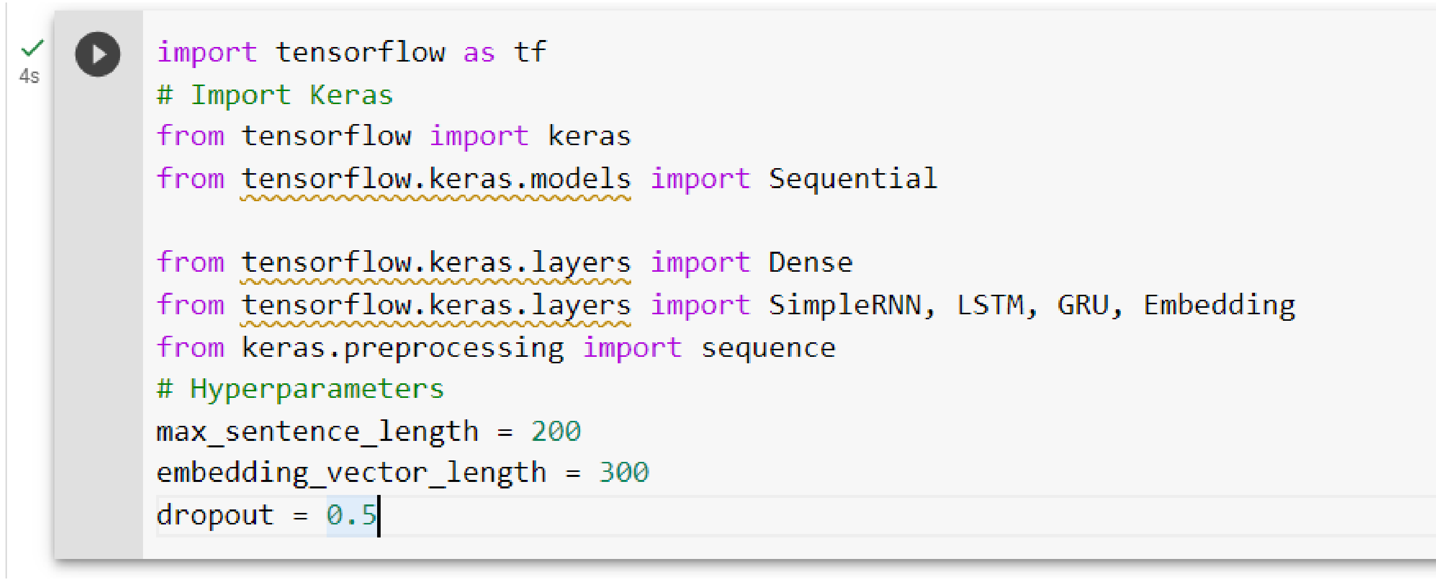
Once the encoder is fitted, I applied it to y_train and y_test as below. We also convert the resulting representations into arrays, as this is the format the Keras will require further down.



```
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()
```

Finally, model definition and training

It's time to train our RNN! So, I imported the packages that are tensorflow, keras that I need and specified some hyperparameters



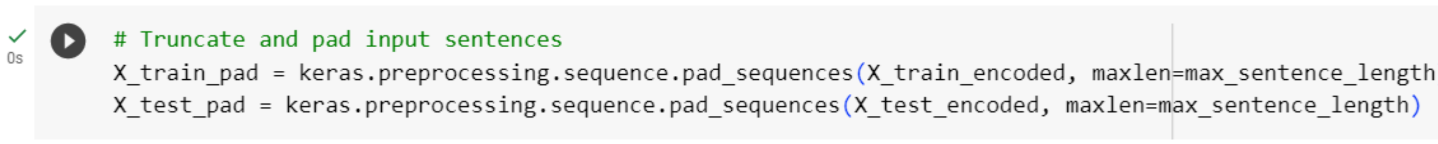
```
import tensorflow as tf
# Import Keras
from tensorflow import keras
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import SimpleRNN, LSTM, GRU, Embedding
from keras.preprocessing import sequence
# Hyperparameters
max_sentence_length = 200
embedding_vector_length = 300
dropout = 0.5
```

max_sentence_length is where i set the limit to 200 which basically means that maximum length of sentences (or text sequences) to 200 tokens. Any sentences longer than this limit will be truncated, and shorter sentences will be padded with zeros to match the maximum length.   (chatgpt, 2022)

Next for embedding_vector_length being equivalent to 300 meaning each word will be represented as a 300-dimensional vector.

Here I also used dropout = 0.5 that is 50% of the neurons in certain layers will be randomly deactivated to reduce overfitting.

Once we have specified the sequence length, we need to pad our data to that length.



```
# Truncate and pad input sentences
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)
```

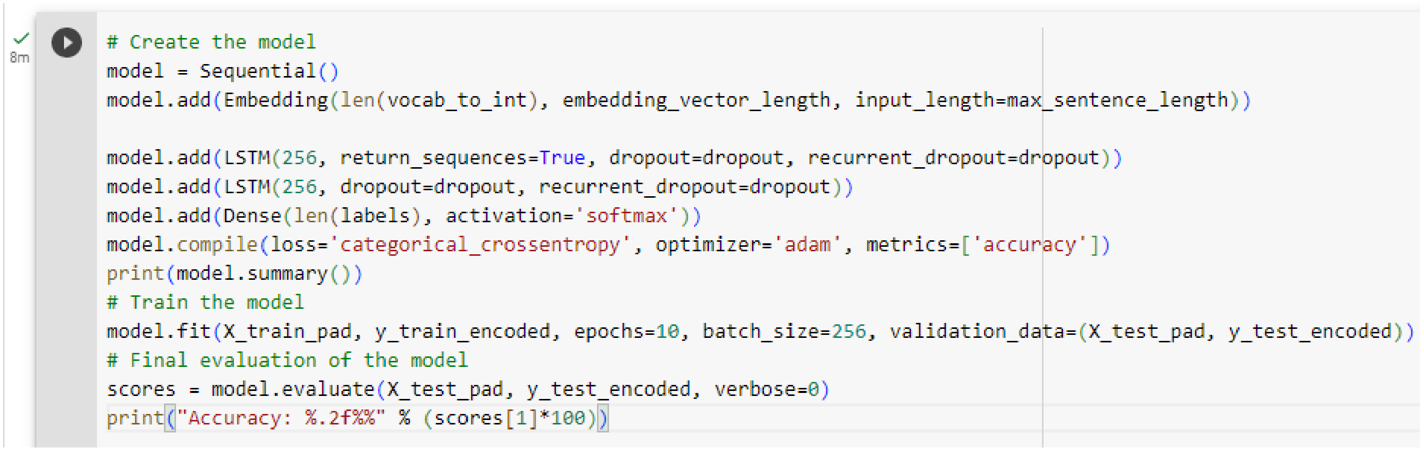
X_train_encoded and X_test_encoded are encoded versions of the input sentences for training and testing, respectively. So, converting text into numerical form that can be processed by the model.

keras.preprocessing.sequence.pad_sequences is a function from the Keras library that performs the truncation and padding of sequences.

X_train_pad and X_test_pad store the padded and truncated versions of the encoded sentences for training and testing, respectively. These padded sequences are suitable for feeding into the sentiment analysis model, which typically requires fixed-length inputs.]

(chatgpt, 2022) (chatgpt, 2022)

a sequence that is shorter than the max sentence length parameters, gets padded up with a random symbol, e.g. -1 or 0 or “PAD”, tensorflow will do this for us automatically.



```
# Create the model
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())
# Train the model
model.fit(X_train_pad, y_train_encoded, epochs=10, batch_size=256, validation_data=(X_test_pad, y_test_encoded))
# Final evaluation of the model
scores = model.evaluate(X_test_pad, y_test_encoded, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

The model is created with an embedding layer followed by two LSTM layers and a dense output layer with softmax activation.

The First layer I added was LSTM (Long Short-Term Memory) layer to the model. To process the sequential data, it had the following parameters:

256: The number of units (neurons/dimensionality) in the LSTM layer.

“return_sequences=True” will return the full sequence of outputs with each input when putting LSTM Layers on top of each other.

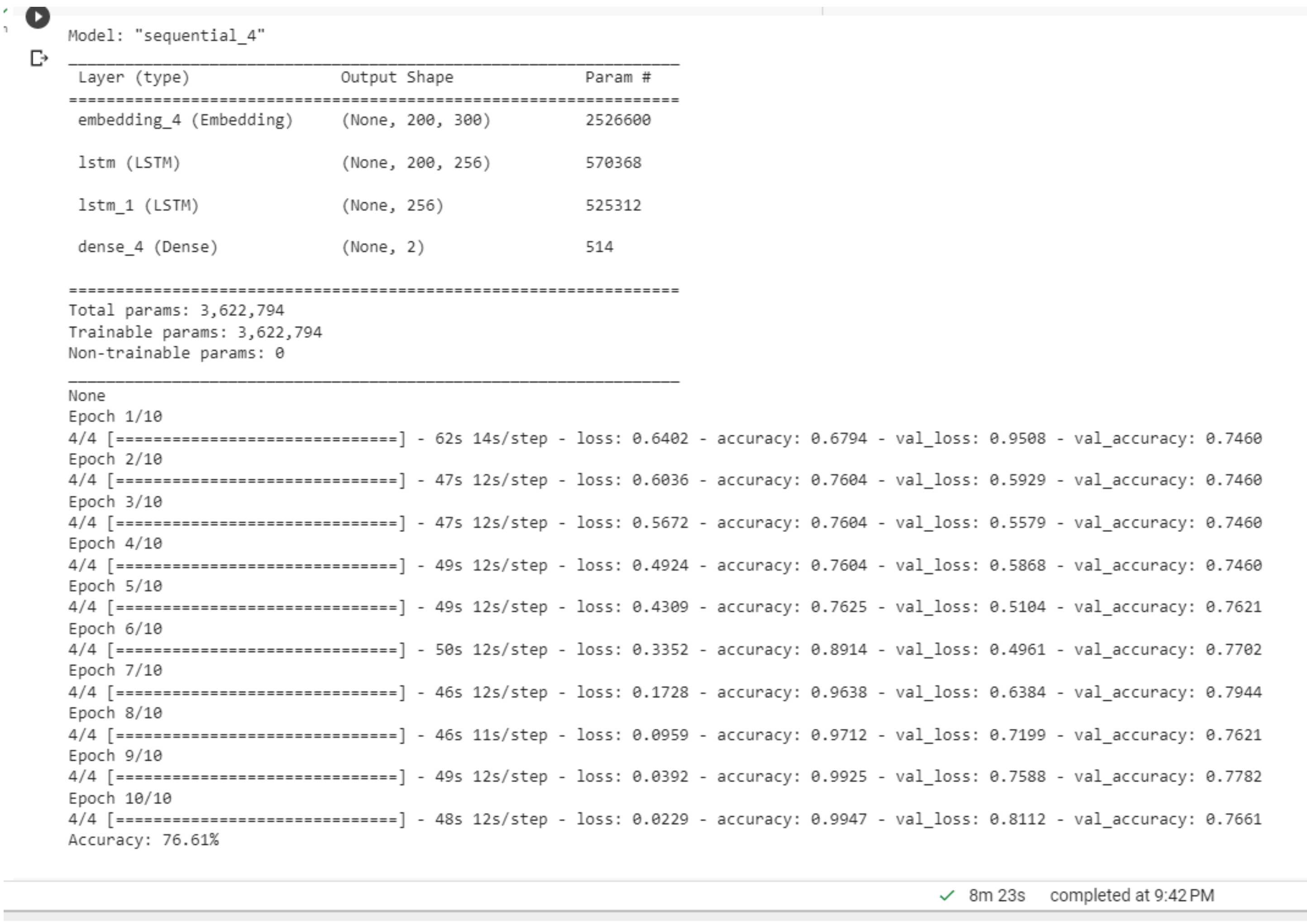
dropout=dropout, recurrent_dropout=dropout: These are the dropout rates for the LSTM layer, which were set earlier as 0.5.

model.add(LSTM(256, dropout=dropout, recurrent_dropout=dropout))- This line adds the second LSTM layer to the model, which is similar to the first LSTM layer but without the return_sequences=True argument, so it will not return the full sequence of outputs.

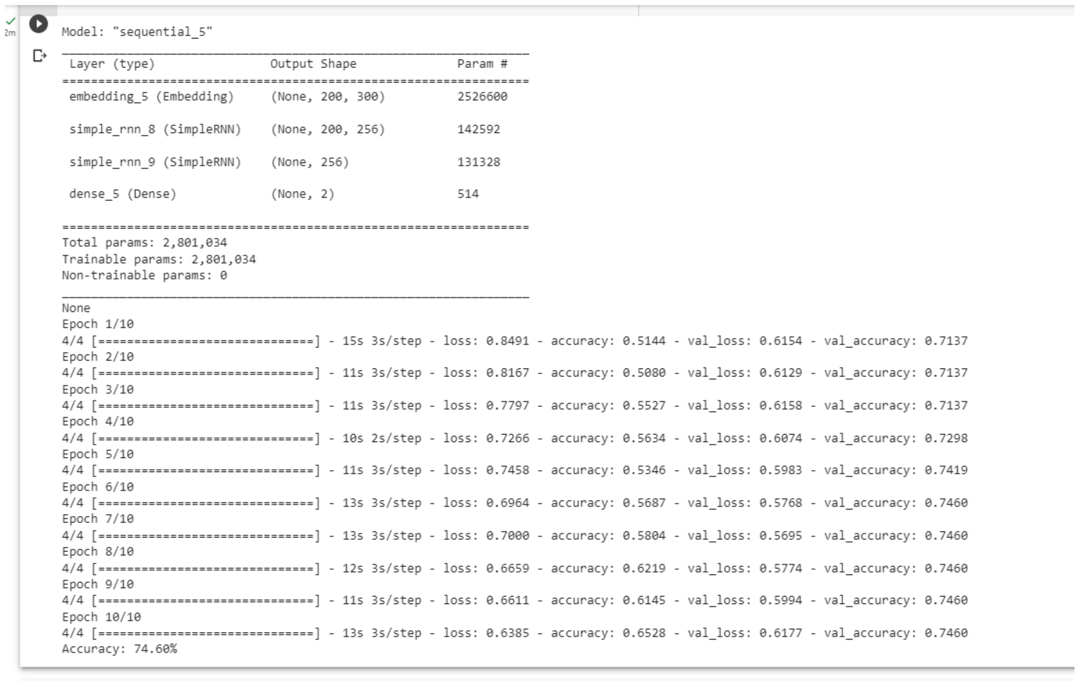
model.add(Dense(len(labels), activation='softmax'))- adds a Dense layer to the model. The Dense layer is a fully connected neural network layer that takes len(labels) number of units (neurons) and uses the softmax activation function. The len(labels) represents the number of classes or unique labels in the target variable.

Results obtained in a table

Below the result for LSTM(the one I chose)



```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_4 (Embedding)     (None, 200, 300)          2526600

 lstm (LSTM)                 (None, 200, 256)          570368

 lstm_1 (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 [==============================] - 62s 14s/step - loss: 0.6402 - accuracy: 0.6794 - val_loss: 0.9508 - val_accuracy: 0.7460
Epoch 2/10
4/4 [==============================] - 47s 12s/step - loss: 0.6036 - accuracy: 0.7604 - val_loss: 0.5929 - val_accuracy: 0.7460
Epoch 3/10
4/4 [==============================] - 47s 12s/step - loss: 0.5672 - accuracy: 0.7604 - val_loss: 0.5579 - val_accuracy: 0.7460
Epoch 4/10
4/4 [==============================] - 49s 12s/step - loss: 0.4924 - accuracy: 0.7604 - val_loss: 0.5868 - val_accuracy: 0.7460
Epoch 5/10
4/4 [==============================] - 49s 12s/step - loss: 0.4309 - accuracy: 0.7625 - val_loss: 0.5104 - val_accuracy: 0.7621
Epoch 6/10
4/4 [==============================] - 50s 12s/step - loss: 0.3352 - accuracy: 0.8914 - val_loss: 0.4961 - val_accuracy: 0.7702
Epoch 7/10
4/4 [==============================] - 46s 12s/step - loss: 0.1728 - accuracy: 0.9638 - val_loss: 0.6384 - val_accuracy: 0.7944
Epoch 8/10
4/4 [==============================] - 46s 11s/step - loss: 0.0959 - accuracy: 0.9712 - val_loss: 0.7199 - val_accuracy: 0.7621
Epoch 9/10
4/4 [==============================] - 49s 12s/step - loss: 0.0392 - accuracy: 0.9925 - val_loss: 0.7588 - val_accuracy: 0.7782
Epoch 10/10
4/4 [==============================] - 48s 12s/step - loss: 0.0229 - accuracy: 0.9947 - val_loss: 0.8112 - val_accuracy: 0.7661
Accuracy: 76.61%
```



```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_5 (Embedding)     (None, 200, 300)          2526600

 simple_rnn_8 (SimpleRNN)    (None, 200, 256)          142592

 simple_rnn_9 (SimpleRNN)    (None, 256)               131328

 dense_5 (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: 0.8491 - accuracy: 0.5144 - val_loss: 0.6154 - val_accuracy: 0.7137
Epoch 2/10
4/4 [==============================] - 11s 3s/step - loss: 0.8167 - accuracy: 0.5080 - val_loss: 0.6129 - val_accuracy: 0.7137
Epoch 3/10
4/4 [==============================] - 11s 3s/step - loss: 0.7797 - accuracy: 0.5527 - val_loss: 0.6158 - val_accuracy: 0.7137
Epoch 4/10
4/4 [==============================] - 10s 2s/step - loss: 0.7266 - accuracy: 0.5634 - val_loss: 0.6074 - val_accuracy: 0.7298
Epoch 5/10
4/4 [==============================] - 11s 3s/step - loss: 0.7458 - accuracy: 0.5346 - val_loss: 0.5983 - val_accuracy: 0.7419
Epoch 6/10
4/4 [==============================] - 13s 3s/step - loss: 0.6964 - accuracy: 0.5687 - val_loss: 0.5768 - val_accuracy: 0.7460
Epoch 7/10
4/4 [==============================] - 13s 3s/step - loss: 0.7000 - accuracy: 0.5804 - val_loss: 0.5695 - val_accuracy: 0.7460
Epoch 8/10
4/4 [==============================] - 12s 3s/step - loss: 0.6659 - accuracy: 0.6219 - val_loss: 0.5774 - val_accuracy: 0.7460
Epoch 9/10
4/4 [==============================] - 11s 3s/step - loss: 0.6611 - accuracy: 0.6145 - val_loss: 0.5994 - val_accuracy: 0.7460
Epoch 10/10
4/4 [==============================] - 13s 3s/step - loss: 0.6385 - accuracy: 0.6528 - val_loss: 0.6177 - val_accuracy: 0.7460
Accuracy: 74.60%
```

This is for SimpleRNN that is slower and less accurate as the time taken by it was 15minutes and even its accuracy is 74.60% and one would go for a higher accracy model.

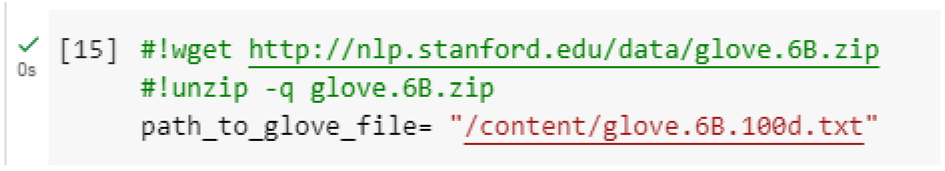
Another reference is from the lab work

PART2: Vary the knowledge representation

Since we already have an in-built layer which is trained based on the word frequencies in the data that the model is provided with (i.e. in our case the deforestation tweets/ texts), we need to add another layer. Apart from the vocab within our dataset which I believe is constrained only with a specific vocabulary I.e deforestation tweets (a specific style of writing),

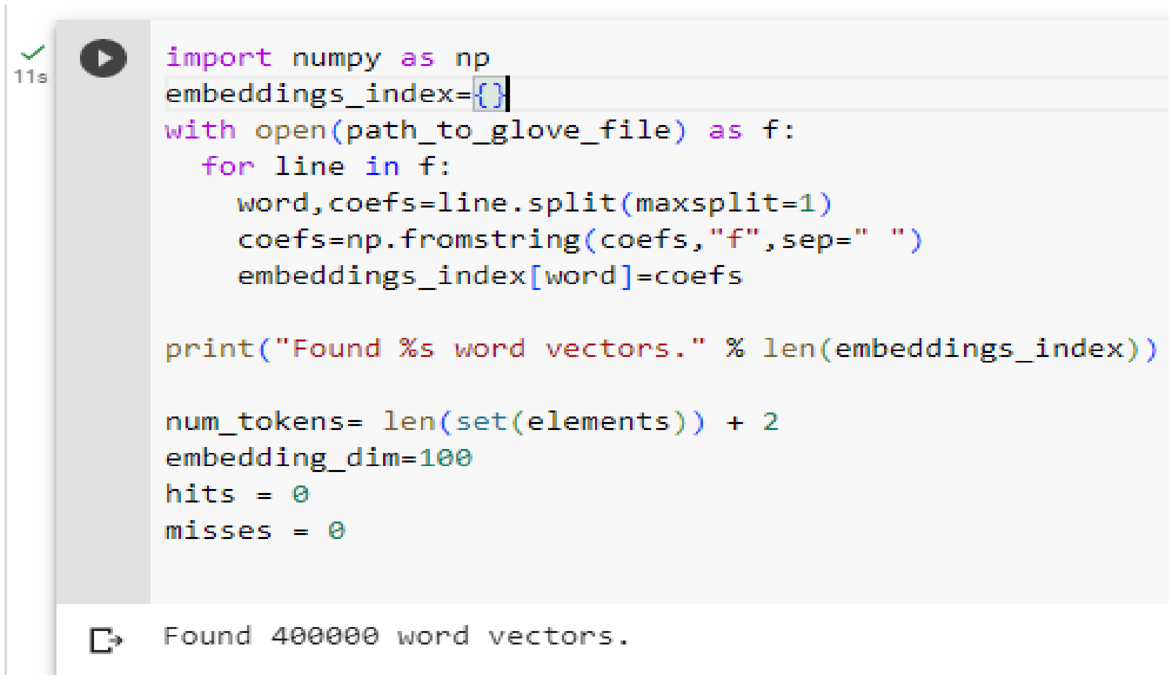
So, if we load Glove embeddings that has in-built "general English" that would help,

So ultimately after having compiled my look-up table of words to integers, I will download and unzip some pre-trained GloVe embeddings



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

I'll be using these embeddings to create an embeddings_index, which will map words
in our vocabulary to their respective embedding representation.



```
import numpy as np
embeddings_index={}
with open(path_to_glove_file) as f:
  for line in f:
    word,coefs=line.split(maxsplit=1)
    coefs=np.fromstring(coefs,"f",sep=" ")
    embeddings_index[word]=coefs

print("Found %s word vectors." % len(embeddings_index))

num_tokens= len(set(elements)) + 2
embedding_dim=100
hits = 0
misses = 0
```

```
Found 400000 word vectors.
```

We then create an embedding_matrix from our embedding_index, which converts whatever words
it can into an embedding_matrix and the matrix is then compared to the data in the word_emdeddings and if they're matched , it's recorded as a "hit" and if not then it's recorded as a "miss")
When running this code, I get the following feedback (output):
Vocabulary of our dataset: 216976
Found 400000 word vectors.
Converted 79693 words (137283 misses)

Path_to_glove_file is loaded, which has pre-trained word embeddings in the GloVe format. It then iterates over each line one after the other. Each line has two parts that are the word and coefs (coefficients). The coefs part(word embeddings) is converted into a NumPy array and the word and its corresponding coefs get added to the embeddings_index dictionary. After processing all lines, the code prints the number of word vectors found in the embeddings_index dictionary.

Here Overall, using GloVe embeddings can offer a good balance of performance, memory efficiency, and ease of use in a wide range of NLP tasks.

Balancing Ethical Concerns and Benefits:

Overall, Researchers should work on Accessibility and Control, Privacy and Data Security, Misinformation and Fake Content, Bias and Fairness.

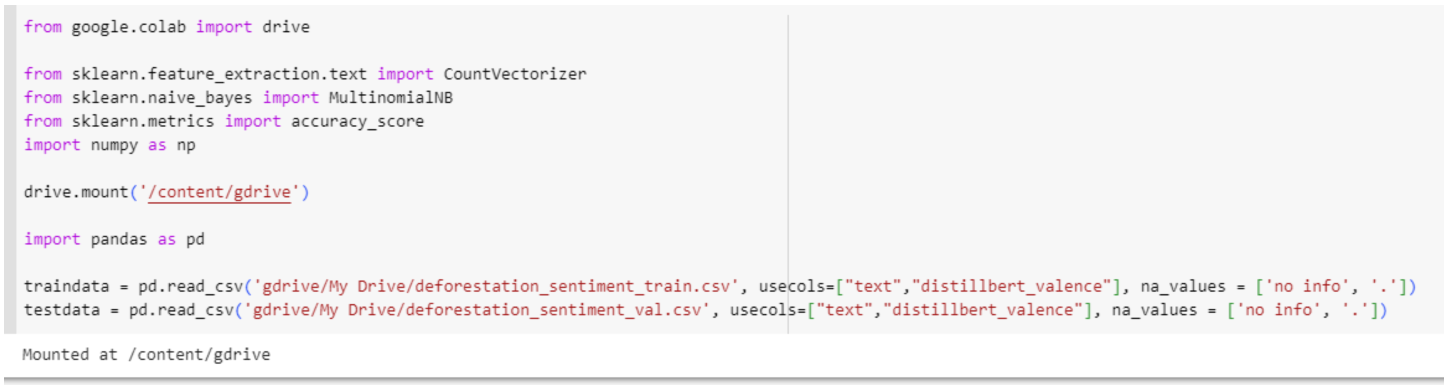
To address the ethical implications, researchers and developers should aim to target main issues such as biases in training data across internet that literally leads to uninformed decisions.

By spreading and promoting fairness and improving transparency in model decision-making, it will be beneficial for us. Efforts need to be made to stop fake content generated by language models are essential otherwise it will lead to a lot of loss Ensuring user privacy and data security should be a priority.

Moreover, the AI community and policymakers should make a big change in the way to train pre -trained language models in a way that it can never be used in any unethical manner by the general society. By making specific guidelines it could be done.

By being aware of the ethical implications and taking proactive steps to address them, the AI community can maximize the positive impact of pre-trained language models while minimizing potential harms.

3. Create a probabilistic baseline



```
from google.colab import drive

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score
import numpy as np

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', '.'])
```

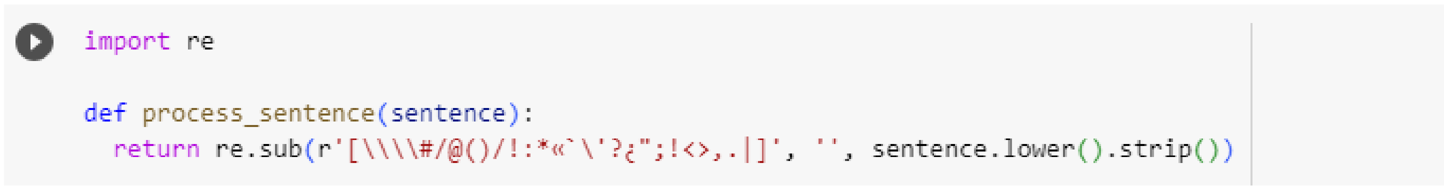
```
Mounted at /content/gdrive
```

I have imported all the necessary libraries for a sentiment analysis task using the Multinomial Naive Bayes classifier.

I have imported the CountVectorizer which is used to convert text data into numerical vectors for training machine learning models.

I have created a dataframe called traindata which is reading the CSV file named "deforestation_sentiment_train.csv" located in the "My Drive" on Google Drive. It selects only the "text" and "distillbert_valence" columns from the CSV file and replaces missing values marked as 'no info' or '.' with NaN .

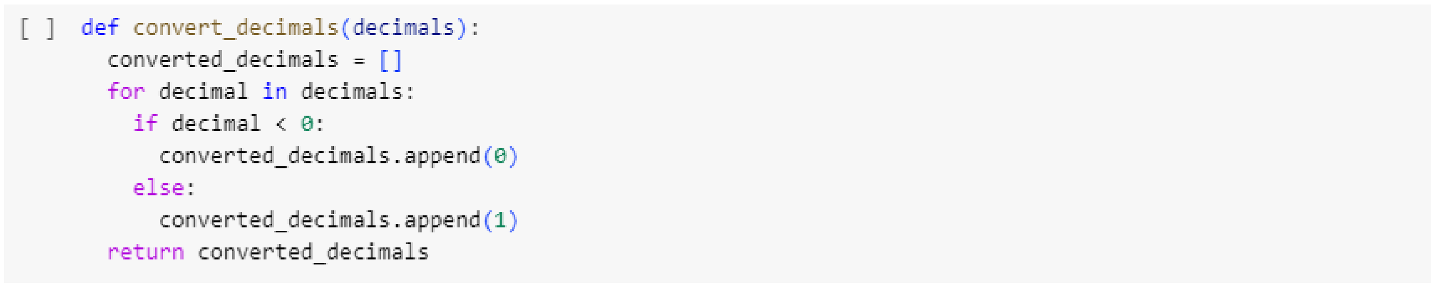
Likewise, I have created a dataframe called testdata which reads the test data from a CSV file named "deforestation_sentiment_val.csv" located in the "My Drive" on Google Drive. It selects only the "text" and "distillbert_valence" columns from the CSV file and replaces missing values marked as 'no info' or '.' with NaN .



```
import re

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

Now for data preprocessing, I created a method process_sentence() that is ultimately converting the lines (tweet sentence) as an input sentence into clean texts as an output by removing all special characters from the sentence.

The re.sub (\\\\V:*«`\'?¿";!<>,.|) will return an output string by replacing all the special characters in the input sentence with an empty string, effectively removing them from the sentence in return.



```
[ ] def convert_decimals(decimals):
      converted_decimals = []
      for decimal in decimals:
        if decimal < 0:
          converted_decimals.append(0)
        else:
          converted_decimals.append(1)
      return converted_decimals
```

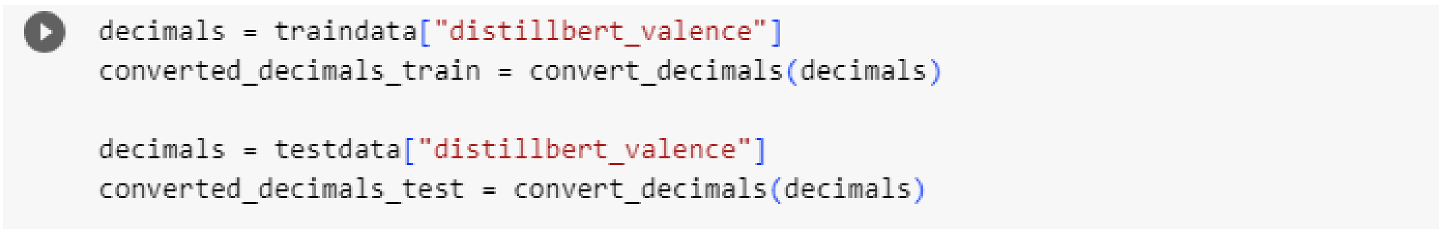
I created a function called convert_decimals that takes one parameter decimals and its purpose is to convert a list of decimal numbers into a list of strings based on their sign positive or negative. An empty list called converted_decimals will store the converted values.

“for decimal in decimals” basically runs off a for loop to go thru each element decimal number in input.

if decimal < 0 -if it’s negative, the string "negative" gets added to the converted_decimals list using the append() method in “converted_decimals.append("negative")”.

Else - If the condition is not true then it goes to the next else block. In the else block, the code adds the string "positive" to the converted_decimals list using the method “converted_decimals.append("positive") -”.

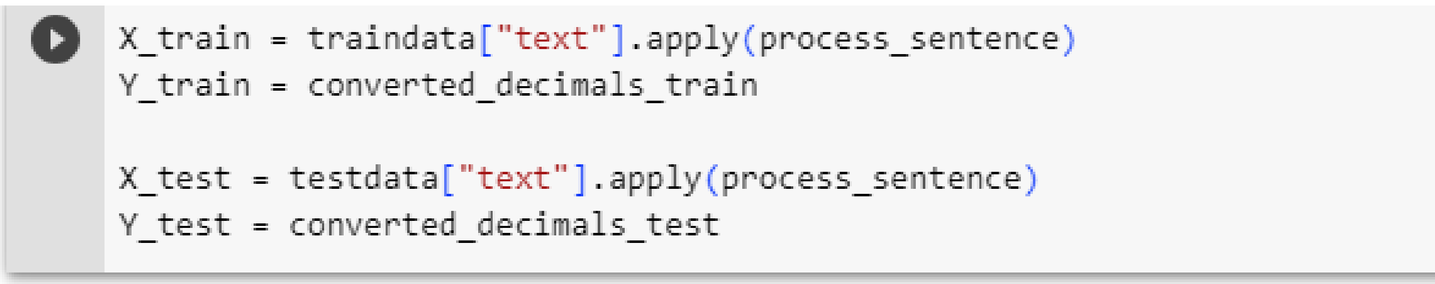
At Last, function reaches the return statement.return converted_decimals as “positive” or “negative” depending on the sign of the corresponding decimal value in the input list .



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

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

convert_decimals function is used to assign these numerical sentiment numbers to categorical labels positive /negative by iterating through each decimal value

“ converted_decimals_train “ has categorical labels for the training dataset, and converted_decimals_test has categorical labels for the test dataset. These categorical labels can then be used as target variables for sentiment analysis or any other classification task.



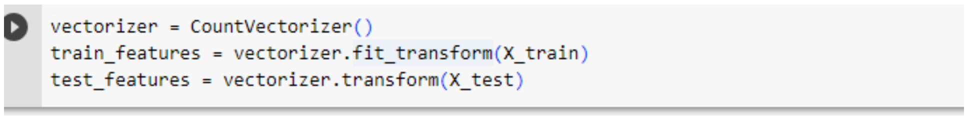
```
X_train = traindata["text"].apply(process_sentence)
Y_train = converted_decimals_train

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

I created a new DataFrame called X_train .I tried to apply the function process_sentence to the "text" column of the DataFrame traindata. The apply() method is applying the process_sentence to each element of the "text" column, and the output is stored in X_train. The process_sentence function cleans the text as explained above.

Another dataframe that I created called ”y_train” is calling the function convert_decimals_train.

Same thing goes on for X-test and Y_test below.

All the converted positive and negative strings that are in “y_train” are stored in labels.



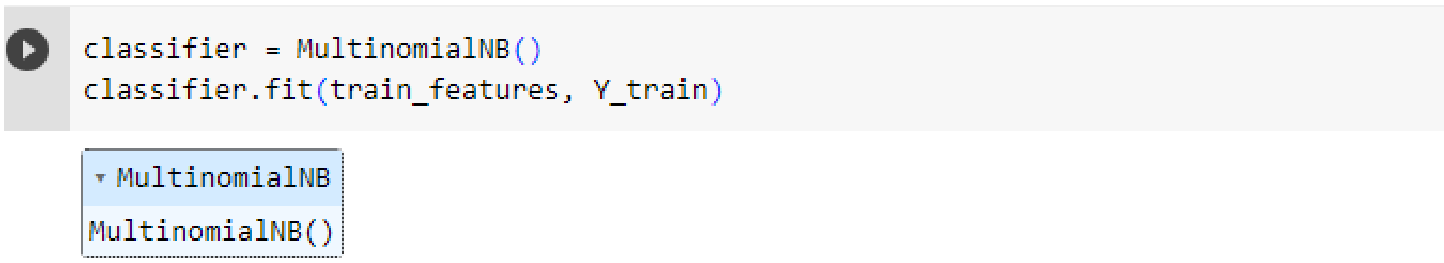
```
vectorizer = CountVectorizer()
train_features = vectorizer.fit_transform(X_train)
test_features = vectorizer.transform(X_test)
```

"sklearn.feature_extraction.text.CountVectorizer - scikit-learn." https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html.[1]

Here CountVectorizer converts a collection of text documents into a matrix of token (word) counts.

train_features = vectorizer.fit_transform(X_train) is a method ie. used to fit the vectorizer on the training data (X_train) and then transform it into the token counts. The fit_transform method learns the vocabulary (unique words) from the training data.

test_features = vectorizer.transform(X_test): The transform method of the CountVectorizer is used to transform the test data (X_test) into the matrix using the same vocabulary learned from the training data. This method is just a way to mix and match same set of words are used for both the training and test datasets.



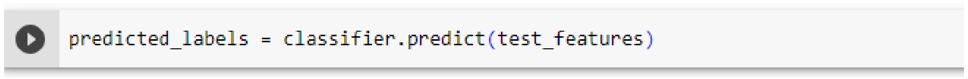
```
classifier = MultinomialNB()
classifier.fit(train_features, Y_train)
```

```
▾ MultinomialNB
MultinomialNB()
```

Classier is trained and is used and will make predictions on new text data that is trained from Multinomial Naive Bayes model. classifier.fit(train_features, Y_train): It’s training the model in short. It has two main arguments:

train_features: The matrix from the CountVectorizer, contains the token counts of the training data. Each row represents a text sample, and each column has its respective unique word in the vocabulary.

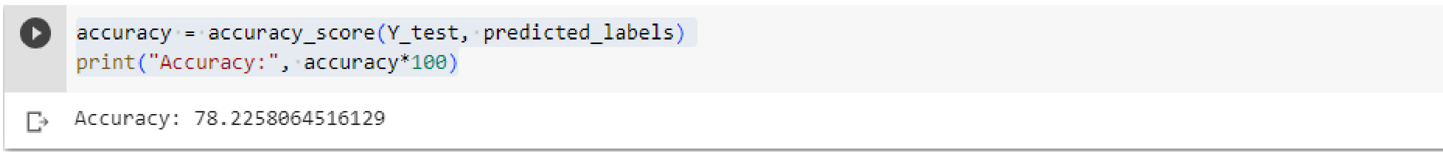
Y_train: The target labels (or sentiment labels) corresponding to each text sample in the training data. These are the categorical labels using the convert_decimals function.



```
predicted_labels = classifier.predict(test_features)
```

1

It’s just predicting the array of the prediticted sentiment analysis that it leanrd from the training phase and test_features is input.



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

```
Accuracy: 78.2258064516129
```

Here accuracy variable is just evaluating how accurate/precise the sentiment anlaysis model.

REFERENCES:

1] Bibliography

chatgpt, 2022. [Online]
Available at: https://chat.openai.com/

Anonymous, 2020. Image Processing. [Online]
Available at: http://www.math.buffalo.edu/~badzioch/MTH337/PT/PT-image_processing/PT-image_processing.html

[Accessed 4 DECEMBER 2022].
Anonymous, -. Embedding layer. [Online]
Available at: https://keras.io/api/layers/core_layers/embedding/
[Accessed 05 dec 2022].
Anonymous, n.d. What is Sentiment Analysis?. [Online]
Available at: https://aws.amazon.com/what-is/sentiment-analysis/
[Accessed 4 DEC 2022].
Baheti, P., 2022. A Simple Guide to Data Preprocessing in Machine Learning. [Online]
Available at: https://www.v7labs.com/blog/data-preprocessing-guide
[Accessed 04 Dec 2022].
Jeffrey Pennington, R. S. C. D. M., 2008. GloVe: Global Vectors for Word Representation. [Online]
Available at: https://nlp.stanford.edu/projects/glove/
[Accessed 05 DEC 2022].