```
!pip install pandas
!pip install keras
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (2.0.3)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas) (2.
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2023.4)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2024.1)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from pandas) (1.25.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->p
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-packages (2.15.0)
```

```
import pandas as pd
from sklearn.model_selection import train_test_split
import string
from string import digits
import re
from sklearn.utils import shuffle
from keras.preprocessing.sequence import pad_sequences
from keras.layers import LSTM, Input, Dense,Embedding
from keras.models import Model,load_model
from keras.utils import plot_model
from keras.preprocessing.text import one_hot
from keras.preprocessing.text import Tokenizer
from keras.models import model_from_json
import pickle as pkl
import numpy as np
```

```
with open('mar.txt','r') as f:
  data = f.read()
# len(data)
```

```
# we need to clean the data
uncleaned_data_list = data.split('\n')
print(len(uncleaned_data_list))

uncleaned_data_list = uncleaned_data_list[:38695]
print(len(uncleaned_data_list))

english_word = []
marathi_word = []
# cleaned_data_list = []
for word in uncleaned_data_list:
  english_word.append(word.split('\t')[:-1][0])
  marathi_word.append(word.split('\t')[:-1][1])

print(len(english_word), len(marathi_word))
```

```
46996
38695
38695 38695
```

```
language_data = pd.DataFrame(columns=['English','Marathi'])
language_data['English'] = english_word
language_data['Marathi'] = marathi_word

# saving to csv
language_data.to_csv('language_data.csv', index=False)

# loading data from csv
language_data = pd.read_csv('language_data.csv')
```

```
language_data.head()
```

	English	Marathi
0	Go.	जा.
1	Run!	पळ!
2	Run!	धाव!
3	Run!	पळा!
4	Run!	धावा!

Next steps: View recommended plots

```
language_data.tail()
```

	English	Marathi
38690	I don't see your name on the list.	मला तुमचं नाव यादीत दिसत नाहीये.
38691	I don't see your name on the list.	मला तुमचं नाव यादीत दिसत नाही.
38692	I don't think that was your fault.	तुझी चूक होती असं मला वाटत नाही.
38693	I don't think that was your fault.	तुमची चूक होती असं मला वाटत नाही.
38694	I don't understand English at all.	मला इंग्रजी अजिबात समजत नाही.

```
#text preprocessing which includes some basic things like lowercase the text, removing punctuation, removing digits, a
english_text = language_data['English'].values
marathi_text = language_data['Marathi'].values

english_text[0], marathi_text[0]

#lowercasing the setences
english_text_ = [x.lower() for x in english_text]
marathi_text_ = [x.lower() for x in marathi_text]

# Text preprocessing
english_text_ = [re.sub("'",'',x) for x in english_text_]
marathi_text_ = [re.sub("'",'',x) for x in marathi_text_]

# remove puntuation
def remove_punc(text_list):
  table = str.maketrans('', '', string.punctuation)
  removed_punc_text = []
  for sent in text_list:
    sentance = [w.translate(table) for w in sent.split(' ')]
    removed_punc_text.append(' '.join(sentance))
  return removed_punc_text
english_text_ = remove_punc(english_text_)
marathi_text_ = remove_punc(marathi_text_)

# removing the digits from english sentances
remove_digits = str.maketrans('', '', digits)
removed_digits_text = []
for sent in english_text_:
  sentance = [w.translate(remove_digits) for w in sent.split(' ')]
  removed_digits_text.append(' '.join(sentance))
english_text_ = removed_digits_text

# removing the digits from the marathi sentances
marathi_text_ = [re.sub("[२३०८१५७९४६]","",x) for x in marathi_text_]
marathi_text_ = [re.sub("[\u200d]","",x) for x in marathi_text_]

# removing the stating and ending whitespaces
english_text_ = [x.strip() for x in english_text_]
marathi_text_ = [x.strip() for x in marathi_text_]



# Putting the start and end words in the marathi sentances
marathi_text_ = ["start " + x + " end" for x in marathi_text_]


#Splitting our dataset

X = english_text_
Y = marathi_text_

X_train, X_test, y_train, y_test = train_test_split(X,Y,test_size = 0.1)

print(len(X_train))

print(len(y_train))

print(len(X_test))

print(len(y_test))
```

```
34825
34825
3870
3870
```

```
def generator_batch(X=X_train, Y=y_train, batch_size=128, tokenizer_input=None, tokenizer_target=None):
    while True:
        for j in range(0, len(X), batch_size):
            encoder_data_input = np.zeros((batch_size, max_length_english), dtype='float32')
            decoder_data_input = np.zeros((batch_size, max_length_marathi), dtype='float32')
            decoder_target_input = np.zeros((batch_size, max_length_marathi, vocab_size_target), dtype='float32')
            for i, (input_text, target_text) in enumerate(zip(X[j:j + batch_size], Y[j:j + batch_size])):
                for t, word in enumerate(input_text.split()):
                    encoder_data_input[i, t] = tokenizer_input.word_index[word]
                for t, word in enumerate(target_text.split()):
                    decoder_data_input[i, t] = tokenizer_target.word_index[word]
                    if t > 0:
                        decoder_target_input[i, t - 1, tokenizer_target.word_index[word]] = 1
            yield ([encoder_data_input, decoder_data_input], decoder_target_input)
```

```
#These tokenizers are essential for converting text data into numerical sequences, which are then used as inputs to yo
from keras.models import Model
from keras.layers import Input, LSTM, Dense, Embedding
import numpy as np

latent_dim = 50

# Initialize and fit tokenizers
input_tokenizer = Tokenizer()
target_tokenizer = Tokenizer()
input_tokenizer.fit_on_texts(X_train)
target_tokenizer.fit_on_texts(y_train)

# Compute vocabulary sizes
vocab_size_input = len(input_tokenizer.word_index) + 1
vocab_size_target = len(target_tokenizer.word_index) + 1

# Compute maximum sequence lengths
max_length_english = max(len(sequence.split()) for sequence in X_train)
max_length_marathi = max(len(sequence.split()) for sequence in y_train)

# Define the input sequences
encoder_inputs = Input(shape=(None,), name="encoder_inputs")
decoder_inputs = Input(shape=(None,), name="decoder_inputs")

# Define the encoder
emb_layer_encoder = Embedding(vocab_size_input, latent_dim, mask_zero=True)(encoder_inputs)
encoder = LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(emb_layer_encoder)
encoder_states = [state_h, state_c]

# Define the decoder
emb_layer_decoder = Embedding(vocab_size_target, latent_dim, mask_zero=True)(decoder_inputs)
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(emb_layer_decoder, initial_state=encoder_states)
decoder_dense = Dense(vocab_size_target, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

# Define the model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])

# Define generator parameters
train_samples = len(X_train)
batch_size = 128
epochs = 50

# Fit the model
model.fit_generator(generator=generator_batch(X_train, y_train, batch_size=batch_size,
                                              tokenizer_input=input_tokenizer,
                                              tokenizer_target=target_tokenizer),
                    steps_per_epoch=train_samples // batch_size,
                    epochs=epochs)
```

```
272/272 [==============================] - 171s 627ms/step - loss: 3.1091 - accuracy: 0.3568
Epoch 28/50
272/272 [==============================] - 170s 625ms/step - loss: 3.0711 - accuracy: 0.3613
Epoch 29/50
272/272 [==============================] - 181s 668ms/step - loss: 3.0345 - accuracy: 0.3652
Epoch 30/50
272/272 [==============================] - 166s 610ms/step - loss: 2.9997 - accuracy: 0.3690
Epoch 31/50
272/272 [==============================] - 166s 611ms/step - loss: 2.9654 - accuracy: 0.3726
Epoch 32/50
272/272 [==============================] - 165s 606ms/step - loss: 2.9340 - accuracy: 0.3762
Epoch 33/50
272/272 [==============================] - 180s 663ms/step - loss: 2.9036 - accuracy: 0.3797
Epoch 34/50
272/272 [==============================] - 167s 616ms/step - loss: 2.8728 - accuracy: 0.3830
Epoch 35/50
272/272 [==============================] - 167s 613ms/step - loss: 2.8430 - accuracy: 0.3865
Epoch 36/50
272/272 [==============================] - 168s 617ms/step - loss: 2.8136 - accuracy: 0.3898
Epoch 37/50
272/272 [==============================] - 179s 658ms/step - loss: 2.7837 - accuracy: 0.3929
Epoch 38/50
272/272 [==============================] - 166s 612ms/step - loss: 2.7548 - accuracy: 0.3963
Epoch 39/50
272/272 [==============================] - 165s 607ms/step - loss: 2.7276 - accuracy: 0.3993
Epoch 40/50
272/272 [==============================] - 179s 658ms/step - loss: 2.6999 - accuracy: 0.4023
Epoch 41/50
272/272 [==============================] - 168s 618ms/step - loss: 2.6728 - accuracy: 0.4058
Epoch 42/50
272/272 [==============================] - 167s 616ms/step - loss: 2.6468 - accuracy: 0.4087
Epoch 43/50
272/272 [==============================] - 168s 619ms/step - loss: 2.6214 - accuracy: 0.4116
Epoch 44/50
272/272 [==============================] - 184s 678ms/step - loss: 2.5957 - accuracy: 0.4146
Epoch 45/50
272/272 [==============================] - 169s 623ms/step - loss: 2.5719 - accuracy: 0.4176
Epoch 46/50
272/272 [==============================] - 170s 626ms/step - loss: 2.5477 - accuracy: 0.4205
Epoch 47/50
272/272 [==============================] - 183s 674ms/step - loss: 2.5231 - accuracy: 0.4230
Epoch 48/50
272/272 [==============================] - 167s 613ms/step - loss: 2.4995 - accuracy: 0.4257
Epoch 49/50
272/272 [==============================] - 168s 615ms/step - loss: 2.4747 - accuracy: 0.4285
Epoch 50/50
272/272 [==============================] - 168s 618ms/step - loss: 2.4516 - accuracy: 0.4312
<keras.src.callbacks.History at 0x7bca7502f700>
```

```
model_json = model.to_json()
with open("model_2.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("model_weight_5.h5")
print("Saved model to disk")

# loading the model architecture and asigning the weights
json_file = open('model_2.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
model_loaded = model_from_json(loaded_model_json)
# load weights into new model
model_loaded.load_weights("model_weight_5.h5")
```

```
Saved model to disk
```

```
latent_dim = 50
#inference encoder
encoder_inputs_inf = model_loaded.input[0] #Trained encoder input layer
encoder_outputs_inf, inf_state_h, inf_state_c = model_loaded.layers[4].output # retoring the encoder lstm output and s
encoder_inf_states = [inf_state_h,inf_state_c]
encoder_model = Model(encoder_inputs_inf,encoder_inf_states)


#inference decoder
# The following tensor will store the state of the previous timestep in the "starting the encoder final time step"
decoder_state_h_input = Input(shape=(latent_dim,)) #becase during training we have set the lstm unit to be of 50
decoder_state_c_input = Input(shape=(latent_dim,))
decoder_state_input = [decoder_state_h_input,decoder_state_c_input]

# # inference decoder input
decoder_input_inf = model_loaded.input[1] #Trained decoder input layer
# decoder_input_inf._name='decoder_input'
decoder_emb_inf = model_loaded.layers[3](decoder_input_inf)
decoder_lstm_inf = model_loaded.layers[5]
decoder_output_inf, decoder_state_h_inf, decoder_state_c_inf = decoder_lstm_inf(decoder_emb_inf, initial_state =decode
decoder_state_inf = [decoder_state_h_inf,decoder_state_c_inf]
#inference dense layer
dense_inf = model_loaded.layers[6]
decoder_output_final = dense_inf(decoder_output_inf)# A dense softmax layer to generate prob dist. over the target voc

decoder_model = Model([decoder_input_inf]+decoder_state_input,[decoder_output_final]+decoder_state_inf)



# Code to predct the input sentences translation
reverse_word_map_target = {v: k for k, v in target_tokenizer.word_index.items()}
def decode_seq(input_seq):
  # print("input_seq=>",input_seq)
  state_values_encoder = encoder_model.predict(input_seq)
  # intialize the target seq with start tag
  target_seq = np.zeros((1,1))
  target_seq[0, 0] = target_tokenizer.word_index['start']
  # print("target_seq:=>",target_seq)
  stop_condition = False
  decoder_sentance = ''
  # print("Beforee the while loop")
  while not stop_condition:
    sample_word , decoder_h,decoder_c= decoder_model.predict([target_seq] + state_values_encoder)
    # print("sample_word: =>",sample_word)
    sample_word_index = np.argmax(sample_word[0,-1,:])
    # print("sample_word_index: ",sample_word_index)
    decoder_word = reverse_word_map_target[sample_word_index]
    decoder_sentance += ' '+ decoder_word
    # print("decoded word:=>",decoder_word)
    # print(len(decoder_sentance))
    # print("len(decoder_sentance) > 70: ",len(decoder_sentance) > 70)
    # print('decoder_word == "end"',decoder_word == 'end')
    # print(decoder_word == 'end' or len(decoder_sentance) > 70)
    # stop condition for the while loop
    if (decoder_word == 'end' or
        len(decoder_sentance) > 70):
        stop_condition = True
        # print("from if condition")
    # target_seq = np.zeros((1,1))
    target_seq[0, 0] = sample_word_index
    # print(target_seq)
    state_values_encoder = [decoder_h,decoder_c]
  return decoder_sentance
```

```
for i in range(30):
  sentance = X_test[i]
  original_target = y_test[i]
  input_seq = input_tokenizer.texts_to_sequences([sentance])
  pad_sequence = pad_sequences(input_seq, maxlen= 30,padding='post')
```