import nltk

from nltk.stem import WordNetLemmatizer

import json

import pickle

import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout

from tensorflow.keras.optimizers import SGD

import random

from tensorflow.keras.models import load\_model

from tensorflow.python.util import deprecation # To remove tensorflow deprecation warnings

import warnings

deprecation.\_PRINT\_DEPRECATION\_WARNINGS = False

warnings.simplefilter(action='ignore', category=FutureWarning)

warnings.simplefilter(action='ignore', category=DeprecationWarning)

lemmatizer = WordNetLemmatizer()

class modelTrain:

def \_\_init\_\_(self):

self.words = []

self.classes = []

self.documents = []

self.ignore\_words = ['?', '!']

def loadIntents(self, intents\_path=''):

data\_file = open(intents\_path).read()

intents = json.loads(data\_file)

return intents

def preprocess\_save\_Data(self, intents):

for intent in intents['intents']:

for pattern in intent['patterns']:

# tokenize each word

w = nltk.word\_tokenize(pattern)

self.words.extend(w)

# add documents in the corpus

self.documents.append((w, intent['tag']))

# add to our classes list

if intent['tag'] not in self.classes:

self.classes.append(intent['tag'])

# lemmatize and lower each word and remove duplicates

self.words = [lemmatizer.lemmatize(w.lower()) for w in self.words if w not in self.ignore\_words]

self.words = sorted(list(set(self.words)))

# sort classes

self.classes = sorted(list(set(self.classes)))

# documents = combination between patterns and intents

print(len(self.documents), " documents ")

# classes = intents

print(len(self.classes), " classes ", self.classes)

# words = all words, vocabulary

print(len(self.words), " unique lemmatized words ", self.words)

# Save data

pickle.dump(self.words, open('words.pkl', 'wb'))

pickle.dump(self.classes, open('classes.pkl', 'wb'))

return self.words, self.classes

def prepareTrainingData(self, words, classes):

# create training data

training = []

# empty output array

output\_empty = [0] \* len(classes)

# training set, bag of words for each sentence

for doc in self.documents:

# initialize our bag of words

bag = []

# list of tokenized words for the pattern

pattern\_words = doc[0]

# lemmatize each word - create base word, in attempt to represent related words

pattern\_words = [lemmatizer.lemmatize(word.lower()) for word in pattern\_words]

# create our bag of words array with 1, if word match found in current pattern

for w in words:

bag.append(1) if w in pattern\_words else bag.append(0)

# output is a '0' for each tag and '1' for current tag (for each pattern)

output\_row = list(output\_empty)

output\_row[classes.index(doc[1])] = 1

training.append([bag, output\_row])

# shuffle our features and turn into np.array

random.shuffle(training)

training = np.array(training)

# create train and test lists. X - patterns, Y - intents

train\_x = list(training[:, 0])

train\_y = list(training[:, 1])

# print("Training data created")

return train\_x, train\_y

def createModel(self, train\_x, train\_y, epochs=200, batch\_size=5, save\_path='model.model'):

"""

Creates a model.

Parameters:

train\_x --> x train values

train\_y --> y train values

epochs --> no of epochs to be trained

batch\_size --> batch\_size during training

save\_path --> path to save the created model

Model Structure:

Layer 1 - 128 neurons, 'relu' activation

Layer 2 - 64 neurons, 'relu' activation

Layer 3 - (no. of classes), 'softmax' activation

Optimizer - Stochastic Gradient Descent --> (best for this example)

learning rate: 0.01

momentum: 0.9

nesterov accelerated --> True

Loss: categorical crossentropy

"""

model = Sequential()

model.add(Dense(128, input\_shape=(len(train\_x[0]),), activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(64, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(len(train\_y[0]), activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

model.compile(loss='categorical\_crossentropy', optimizer=sgd, metrics=['accuracy'])

# fitting and saving the model

hist = model.fit(np.array(train\_x), np.array(train\_y), epochs=epochs, batch\_size=batch\_size, verbose=1)

model.save(save\_path, hist)

print("Model Successfully Created and saved")

return model

class modelPredict:

def \_\_init\_\_(self, intents\_path='filename.json', model\_path='model\_name.json'):

self.intents\_path = intents\_path

self.model = model\_path

def clean\_up\_sentence(self, sentence):

lemmatizer = WordNetLemmatizer()

# tokenize the pattern - split words into array

sentence\_words = nltk.word\_tokenize(sentence)

# stem each word - create short form for word

sentence\_words = [lemmatizer.lemmatize(word.lower()) for word in sentence\_words]

return sentence\_words

# return bag of words array: 0 or 1 for each word in the bag that exists in the sentence

def bow(self, sentence, words, show\_details=False):

# tokenize the pattern

sentence\_words = self.clean\_up\_sentence(sentence)

# bag of words - matrix of N words, vocabulary matrix

bag = [0] \* len(words)

for s in sentence\_words:

for i, w in enumerate(words):

if w == s:

# assign 1 if current word is in the vocabulary position

bag[i] = 1

if show\_details:

print("Found in bag: %s" % w)

return np.array(bag)

def predict\_class(self, sentence, model, error\_threshold=0.25):

ERROR\_THRESHOLD = error\_threshold

words = pickle.load(open('words.pkl', 'rb'))

classes = pickle.load(open('classes.pkl', 'rb'))

# filter out predictions below a threshold

p = self.bow(sentence, words, show\_details=False)

res = model.predict(np.array([p]))[0]

# ERROR\_THRESHOLD = 0.25

results = [[i, r] for i, r in enumerate(res) if r > ERROR\_THRESHOLD]

# sort by strength of probability

results.sort(key=lambda x: x[1], reverse=True)

return\_list = []

for r in results:

return\_list.append({"intent": classes[r[0]], "probability": str(r[1])})

return return\_list

def getResponse(self, ints, intents\_json):

import random

tag = ints[0]['intent']

list\_of\_intents = intents\_json['intents']

for i in list\_of\_intents:

if i['tag'] == tag:

result = random.choice(i['responses'])

break

return result

def chatbot\_response(self, msg):

"""

Outputs a response from the model.

Pass in the input text to receive a response from the model.

Parameters:

msg --> The required input text

"""

model = load\_model(self.model)

intents = json.loads(open(self.intents\_path).read())

ints = self.predict\_class(msg, model)

res = self.getResponse(ints, intents)

return res

# response\_from\_bot = chatbot\_response(input\_query)