d:\Work_folders\code\python-Spyder\Shravani-Chatbot\app.py

```
#%%
import os
from flask import Flask, render_template, url_for, request, jsonify, session
import time
import warnings
warnings.filterwarnings("ignore")
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
#tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
import dialogue
import pandas as pd
#%%
print(os.getcwd())
app = Flask(__name__)
global chat
chat=dialogue.chat_interface_handler()
#%%
######################################################################
# redirect - used to call the decorated function with appropriate parameters passed.
#####################################################################################
# Function rendering simple template
#####################################################################################
@app.route('/')
def home() :
    return render_template("home.html")

#####################################################################################
# End of Function
#####################################################################################
#####################################################################################
# Function accepting simple post and examining it to produce output
#####################################################################################
@app.route("/uin",methods=["POST"])
def user_requests():
    """User input is handled as an XML file from the POST mechanism"""
    print("Request : recieved")
    if request.method == "POST":
        print("Identified as request : POST")
        #print(type(request))
        #print(str(request))
        print("Request is :",request)
        try:
```

```
            #print("Request Arguments :",request.args)
            #print("Request form Keys :",request.form.keys())
            inp=request.form["data"]
            txt=chat.dialog_in(inp)
            return txt
        except:
            print("This has failed")
            return "There is something wrong with Data"
        #print("received packets as it is")
        #print(uinput.xml")


##########################################################################
# End of Function
##########################################################################

##########################################################################
# Function accepting simple close request
##########################################################################
@app.route("/close",methods=["POST"])
def close():
    """Closes all the functions """
    pass
##########################################################################
#
##########################################################################

##########################################################################
#
##########################################################################
#
##########################################################################
#%%

##########################################################################
# Running the Basic Flask app here
##########################################################################
if __name__=='__main__':
    app.run()
```

```
# -*- coding: utf-8 -*-
"""
This file contains the Dialogue Management Enginer, the first layer behind the Core Flask application
@Sheshank Joshi

NOTE : Spelling handling and the information is not yet handled. It is delegated to the language model.
However, spelling mistakes in yes or no options is not entertained, though the case-sensitive issue
can be eliminated.
NOTE : Delete method yet to be implemented that will log the data that is being generated here for
future training purposes.
NOTE : Flask handle will destroy the existing object and create a new object.
"""
#%%
import random
import json
import model_management_engine as me
import copy
#%% Cell

class chat_interface_handler():
    # Don't forget to write functions for data preparations and language interfacing.
    _engine=None
    def __init__(self):
        """The state-machine has three states. 1. Dialogue is yet to be input 2. Dialogue input has been given
        but the score is not appropriate to take decision. 3. Dialogue state is satisfied and next input is to be taken"""
        self._engine=me.model_management_engine()
        self._current_desc_state=1 #(This is for debugging purposes only) #1 # Description state analyze and send the appropriate data and check on it.
        self.inps=[]
        self.outs=[]
        self.dialogues={} # This is the actual storage of the data where everything else is stored.
        self.current_state=1 #2 #(This is for debugging purposes only) #1 # Indicating the Nodal point of the dialogflow ( Crucial parameter for smooth flow of dialogue)
        self.NN_prediction=None # This prediction is a rough interfacing
        self.text_in=""
        self.score=0 # Indicative of whether the processing of description already done.
        self.target_name=None
        self.responses=None # Set of responses based on the dialogue flow state.
        self.response_state=0
        self.col=None # This is the columns of the original dataset as per requirements. A
```

```
        better mechanism for interfacing is needed.
        self._AL=None # Indicating which column we are considering given we already have text description
        self.return_statement="" # WE have to put default entry level introduction text here. Handled by initializer method.
        self.supervised_entry={} # This is where we collect the actual data for supervised learning model
        self.col_entry={} # This is the dictionary holder for current state in dialog values
        self.supervised_data=self._engine.options # Keys should be strings here.
        self.no_of_cols=len(self.supervised_data.keys())-1
        # Entry is of type parameter with an id and a dictionary associated with possible values.
        self.options_given=None
        self.load_response() # Loading the standard responses from saved file.
    #
    # Printing Welcome message as soon as the machine is initialized is not yet done. Do it.
    #
    #
    def debug(self):
        print("\n-----------DEBUG--------------")
        print("Current Description State :",self._current_desc_state)
        print("Current State of the Chatbot :",self.current_state)
        print("Current Description Score :",self.score)
        print("Current Response state for Description :",self.response_state)
        print("Current Col :",self.col)
        # print("Paramter at hand :",self._parameter)
        print("Current Supervised Entry :",self.supervised_entry)
        print("-----------------------------------")
        print("Options given to the user :\n",self.options_given)
        print("-----------------------------------\n")
    #
    #
    def load_response(self):
        file=open("./model_config/responses.json","r")
        resp=json.load(file)
        self.responses=resp
        file.close()
        f=open("./model_config/NLP_data.json","r")
        AL=json.load(f)
        self._AL=AL
        _=self._AL.pop("best_model")
        self.target_name=list(self._AL.keys())[0]
        self._AL=self._AL[self.target_name]
        print("Succesfully Loaded Responses")
        #print(self._AL)
    #
    #
```

```
    def dialog_in(self,text):
        """This will store the temporary input and based on it will handle what is the current state in the dialog flow and to which
        handler data needs to be output to and return appropriate response from the response methods."""
        #print("Dialog in :",text)
        self.return_statement=""
        #state=self.current_state
        #print("The Number of columns in the Data :",self.no_of_cols)
        #print("Current State :",self.current_state)
        try:
            assert self.current_state<self.no_of_cols+2 # This exception means that dialogue management has finished and prediction is over. Ask user to reset the options
            if text=="reset_chat":
                raise ValueError
            if text=="next_item": # This is the request from client to take the conversation to next step. Its a magic code. Initialized after the page is loaded.
                if self._current_desc_state==1:
                    self.initializer(kind="desc")
                    # Description initializer
                if self._current_desc_state==3:
                    self.initializer(kind="sup")

                    self.return_statement=self.responses["desc_accepted"]+"\n"+self.return_statement
            else:
                self.inps.append(text)
                if self._current_desc_state==3:
                    # This means the description event has been successfully accepted and now we are moving towards supervised model
                    self.supervised_data_handler()
                else :
                    # This means the descritpion event hasn't been finished yet.
                    self.dialogue_handler()
        except:
            txt="I am closing this session. Please Reset to use me again."+"\n"+" Or Let me reset it for you. Just type \"reset_chat\" in the chatbox"
            if text=="reset_chat":
                self._current_desc_state=1
                self.response_state=0
                self.supervised_entry={}
                self.score=0
                self.current_state=1
                self.options_given=None
                txt="Thanks for coming back. Start reporting a new incident again. Describe it"
            # Flask will re-initialize the state of resetting the object and clearing everything. Old object is logged into database.
```

```
            self.return_statement=txt
        else:
            #print("The Number of columns in the Data :",self.no_of_cols)
            #print("Current State :",self.current_state)
            if self.current_state==self.no_of_cols+2:
                predicted_AL=self._engine.predict_AL()
                #print("Predicted Accident Level",predicted_AL)
                #print("Supervised Model Data Collected :",self.supervised_entry)
                self.supervised_entry.update({self.target_name:self._AL[str(predicted_AL)]})
                # This is the crucial moment of the code.
                predicted_PAL=self._engine.predict_sup(self.supervised_entry)
                statement=self.responses["successful_pred"]
                # Your accident level is predicted to be ____ with confidence ____
                statement = predicted_PAL +"\n\n"+ statement
                self.return_statement=statement
        return self.return_statement
    #
    #
    #
    #
    def supervised_data_handler(self):
        """Supervised Data handling should happen with the data points. This handles all the
        supervised Learning model data."""
        entry=self.inps[-1]
        #if self.col==None:
        #    self.col_entry=self.supervised_data[self.current_state]
        #    self.col=list(self.col_entry.keys())
        #print("Entry Received :",entry)
        try:
            self.choice_sup_analyzer(entry)
        except:
            self.return_statement = self.responses["err_sup_output"] #+ "\n\n" + self.outs[-1]
            # Here is the key error
            #
            # "That's not a valid choice. Please on"
            # Should say its not a valid choice. please select proper data
        else:
            # Write code here to generate more options
            #self.current_state+=1
            if self.current_state==self.no_of_cols+2:
                #print("End has been reached")
                pass
            else:
                self.choice_sup_generator()
    # Generates next set of options for the next state.
    #
    #
```

```
    #
    def choice_sup_generator(self):
        """Generating supervised model options based on current dialog flow state."""
        self.col=list(self.col_entry.keys())[0]
        #print("Column selected",self.col)
        #print(self.col_entry[self.col])
        choices=self.col_entry.pop(self.col)
        #print(choices)
        if len(choices)==0:
            #This indicates the end has been reached.
            self.current_state+=1
        text="\n".join([str(key) + " . " + str(value) for key,value in enumerate(choices)])
        self.options_given=text
        statement=self.responses[self.col]
        #print(statement)
        self.return_statement=statement+"\n"+text
        self.outs.append(text)
        #self.dialogues["server"].append(text) #Remove this after debugging.
        #print("Response given :\n",self.return_statement)
    #
    #
    #
    #
    def choice_sup_analyzer(self,entry):
        """ This crucially analyzes if the option input is the valid option or not. If not ↵
↳       sends appropriate responses or mitigates it further"""
        choices=self.supervised_data[self.col]
        state=self.current_state
        #print("-----------INPUT ANALYSIS-----------------")
        for key,value in enumerate(choices):
            if str(key) == entry or str(value) == entry: # Checking if the project
                self.supervised_entry.update({self.col:value}) # One column data entry ↵
↳               update is finished
                self.current_state+=1
                break
            #print("Key :",key,"Value :",value)
        #print("------------------------------------------")
        try:
            assert not (state==self.current_state)
        except:
            raise ValueError
    #
    #
    #
    #
    def initializer(self,kind):
        """This will initialize the interface to face various options for the model under ↵
```

```
        consideration."""
        if kind=="desc":
            try:
                assert self._current_desc_state==1
                self.return_statement=self.responses["welcome_msg"]
            except:
                print("There is some fatal error somewhere")
                pass
        elif kind=="sup":
            try:
                assert self._current_desc_state==3
                assert self.current_state>1
                self.options_data=None
                try:
                    temp=self.supervised_data.pop(self.target_name)
                except:
                    pass
                # This is hard coded here. Take care of this.
                #self._AL = dict([(str(key),value) for key,value in enumerate(temp)])
                self.col_entry=copy.copy(self.supervised_data)
                self.choice_sup_generator()
            except:
                print("There is some fatal error somewhere")
                raise ValueError("There is a fatal Error Somewhere")
        else:
            print("There is a fatal error somewhere while initializing, check out.")
    #
    #
    #
    #
    def dialogue_handler(self):
        """diagloues are designed such that whenever a new input comes in, only the latest
        input is considered while the processing is happening in the background.
        This handles all the dialogues that are input"""
        #print("Entered Dialogue Handler")
        txt=self.inps[-1]
        if self._current_desc_state==1:
            self.dialogues.update({"user":[],"server":[]})
            try:
                self.dialogue_analyze(txt)
                # Write code here in case of a success
                self.return_statement=self.return_statement+self.responses["desc_accepted"]
            except ValueError:
                #print("Error Generated. Description was not sufficient")
                ret_statement=self.lm_error_msg_generator()
                #print(ret_statement)
                self.dialogues["server"].append(ret_statement)
```

```
                self._current_desc_state=2
                self.return_statement=ret_statement
            else:
                self.dialog_in("next_item")
                #make function elaborate in dialogue system while changing states.
        elif self._current_desc_state==2:
            try:
                assert self.response_state<4 # this is a tunable parameter. It can be modified on how many attempts user is allowed to make at describing event.
            except:
                # This means the attempts have been exhaust and the user is inputting some
                # irrelevant data so the dialogues should restart and end. Implementation of that is pending.
                #print("Error ! It seems that you are trying to give a description of event that hasn't occurred or is not according to ")
                #print("this is a debug statement for everything else")
                self.return_statement=self.responses["err_desc_output"] # Ask to reset and try again.
            else:
                self.additional_desc_handler(txt)
    #
    #
    #
    def additional_desc_handler(self,txt):
        """After the first two attempts at getting description failed, this will handle the yes or not option for adding data from the user based on language model inputs"""
        text=txt # Taking in the latest input
        #print("The present response State is :",self.response_state)
        if self.response_state>2:
            self.choice_desc_analyzer(text)
            # A Yes or No Question is being asked here, based on most possible bigram correlation. If no is given as input
        elif self.response_state<=2:
            try:
                self.dialogue_analyze(text) # Only to be called when choice handler fails
                # Way to generate options is not taken
            except ValueError:
                ret_statement=self.lm_error_msg_generator()
                # Script to generate options data.
                #self.dialogues["server"].append(ret_statement)
                #self.response_state+=1
                self.return_statement=ret_statement
            else:
                self.dialog_in("next_item")
                self._current_desc_state+=1
                self.current_state+=1
                #expression to handle if there is no exceptions i.e. details were enough.
```

```
        else:
            print("There is something wrong. This shouldn't be called.")
    #
    #
    #
    #
    def dialogue_analyze(self,txt):
        """The core function that will analyze the dialgoue and decide if there is enough score for given description."""
        #self.dialogues["user"].append(text)
        #print(txt)
        #print("Enter Dialogue Analyze \n------------------------------\n")
        self.text_in=self.text_in+ " " + txt # Storing the description text for future analysis
        score,corp=self._engine.fetch_score(self.text_in) # Languge model score extraction along with obviously cleaned dataset comes out.
        std_score=self._engine.avg_score()
        #print("Score found is :",score)
        #print("Average Score is :",std_score)
        if score>=std_score:# if score is greater than average score
            #print("The corpus received is :",corp)
            #print("The Score for the input found satisfactory is :",score)
            self.NN_prediction=self._engine.predict_rough(corp)
            #print("Rough Prediction is :",self._engine.AL_prediction)
             # Fetching rough score from the engine for go-ahead.
            try :
                assert self.NN_prediction == True # rough prediction for threshold analysis approved or not
                #print("NN Prediction Test passed")
            except:
                raise ValueError
            else:
                self._current_desc_state=3 # Indicating successful description grasp and status change
                self.score=score # Storing the scoring paramters for confidence in future
                self.current_state+=1 # Changing to next node in dialog flow
                self.dialogues["user"].append(self.text_in) # Finalized user input is taken as an input successfully capture at one particular state
        else:
            self._current_desc_state=2 # Marking that more description is needed.
            raise ValueError
    #
    #
    #
    #
    def choice_desc_analyzer(self,txt):
        """Checks if the user has input a choice in terms of 1 or 2 or 3, or, Yes or No or not known, if its not yes or no unknown is default taken
```

```
        and based on that it either returns """
        # Also need a choice creator via lm_error_msg_generator
        # A small text analysis code here to convert to lower code or to convert number to ↵
↳       option
        text="" # lower the cases and everything involved
        try:
            assert self.options_given
        except:
            raise NotImplementedError("The Particular Dialogue state hasn't been reached. ↵
↳           Please check program")
        if not txt.isalpha():
            try:
                assert str(txt) in self.options_given.keys()
                added_data=self.options_given[txt]
                self.dialogue_analyze(added_data)
            except:
                self.return_statement=self.responses["not_a_valid_option"]
                raise ValueError("Not a valid option")
        else:
            # This means one of the options given is not selected.
            try:
                text=txt.lower()
            except:
                print("Invalid option") # Error handler appropriately.
            else:
                if self.response_state==2:
                    if text!="no":
                    # handler for bigram in texts; a yes or no question on bigram check
                        self.dialogue_analyze(text)
                    else: # or "1" is also accepted as possible choice
                        #Jump to asking for trigram and call the error message generator
                        ret_statement=self.lm_error_msg_generator()
                        self.dialogues["server"].append(ret_statement)
                        self.return_statement=ret_statement
                        self.response_state==3
                elif self.response_state==3:
                    if text!="no":
                        # handler for trigram in texts; a yes or no question on trigram check
                        self.dialogue_analyze(text)
                    else:
                        # Case of absolute failure
                        self.response_state=4
                        self.dialogue_handler()
                else:
                    print("There is something wrong here.")
    #
    #
```

```
    #
    #
    def lm_error_msg_generator(self):
        """Language model error handler that handles exclusively language model errors and
        generates dynamic new messages based on current state
        of the chat for present description"""
        #num=random.randint(1,10)
        #print("The present response State is :",self.response_state)
        try:
            assert self._current_desc_state==2
            if self.response_state==0:
                # This is the primary response indicating that user's description wasn't
                enough.
                #self.response_state+=1
                #print("This is reached means first warning that description wasn't enough")
                response_text=self.responses[str(self.response_state)][random.randint(0,9)]
            elif self.response_state==1:
                # simple response of asking for more information giving example descriptions
                randomly
                choices=self.choice_desc_generator(choice=1)
                response_text=self.responses[str(self.response_state)][random.randint(0,9)]
                response_text=response_text+"\n"+choices
                #print("This is reached means second warning and choices are given")
                #self.response_state+=1
            else:
                # This handles user suggestions on frequently occurring bigrams and incident
                mechanisms.
                response_text=self.responses[str(self.response_state)][random.randint(0,4)]
                if self.response_state==2:
                    # Handle the extra tests here with appropriate choice generator method
                    # self.bigram_used=lm.get_bigram(self.text_in) # determine the bigram
                    used # Write code that delegates this to the engine
                    choices=self.choice_desc_generator(choice=2)
                    self.options_given=choices
                    response_text=response_text + "\n" + choices # Use some most frequent
                    bigram words with collocation sentences

                elif self.response_state==3:
                    # self.trigram_used=lm.get_trigram(self.text_in) # determine the trigram
                    used # Write code that delegates this to the engine
                    choices=self.choice_desc_generator(choice=3)
                    self.options_given=choices
                    response_text=response_text + "\n" + choices # Use some most frequent
                    trigram words with collocation sentences
                else:
                    print("Error debug statement. Something is wrong.")
            self.outs.append(response_text) # append it as out
```

```
            #print("\n--------------------\nFrom inside Error Message Generator :")
            #print(response_text)
            #print("-----------------------------------")
            self.response_state+=1
            #print("Changed Response State :",self.response_state)
            return response_text
        except:
            raise ValueError ("funciton is triggered from ")
    #
    #
    #
    #
    def choice_desc_generator(self,choice):
        """Checks if the description given is according to the choices or not after first two options are exhaust"""
         # Check if the code has bigram and trigrams both of them same. If same, move to the next one.
        texts=[]
        if choice == 1:
            texts=self._engine.generate_options(option=1)
        else:
            if choice == 2:
                texts=self._engine.generate_options(option=2)#- Option generating for common contexts
                for i in range(len(texts)):
                    texts[i]=str(i)+ " . " + texts[i]
                 # choice generator for bigram data with third or fourth bigram
                #self.options_given=dict([(key+1,value) for key,value in enumerate(texts)])
            elif choice==3:
                texts=self._engine.generate_options(option=3)#- Option generating for concordance
                # Choice generator for trigram data
                for i in range(len(texts)):
                    texts[i]=str(i)+ " . " + texts[i]
            self.options_given=dict([(key+1,value) for key,value in enumerate(texts)])
        #print("Given options are :",self.options_given)
        return "\n".join(texts)
    #
    #
    #
    #
```

```
# -*- coding: utf-8 -*-
"""
Created on Fri Sep 10 17:44:18 2021
This is the right hand module that has the actual language model initited based on the corpus created.
Yet to finish : The train phase of the language models. It can be mitigated or can be run locally.

NOTE : Also mention the research paper links that is going to state and connect the langauge model
probability distributions given here to the actual models.

NOTE : Training Set is assumed to have no spelling mistakes and is proof read is what is assumed. Any Nouns and Named Entities in
training set are not entertained if not in English language. If wish to use, should have used name tagged dataset and upset the
generalization mechanism.

NOTE TO ME : Have to build a recommendation system.
NOTE : A Destructor method needs to be called to clear all the variables and free the memory for further usage.

NOTE : A User note is required where we need to issue guidelines to the user on how to use the chatbot effectively, with most
intensive and effective usage with best description coming within first 'n' number of words (which is a tunable parameter)
@author: Sheshank_Joshi
"""
#%%
import nltk
# Uncomment below comments while deployment.
nltk.download("stopwords")
nltk.download("punkt")
#nltk.download("words")
nltk.download("wordnet")
import pandas as pd
from nltk.lm.models import KneserNeyInterpolated,MLE
from nltk.lm.preprocessing import padded_everygram_pipeline
from nltk.lm import Vocabulary #NgramCounter,
from nltk.tokenize import word_tokenize
from nltk.text import TextCollection
from nltk.util import flatten,ngrams,pad_sequence,bigrams,trigrams,ngrams
from nltk.corpus import stopwords as sw
from nltk.corpus import wordnet as eng_words
from nltk.stem import WordNetLemmatizer, PorterStemmer
from nltk import ContextIndex
import string
```

```
from dateutil.parser import parse
from string import punctuation as punct
from textblob import Word as suggest
import pickle as pkl
import copy
import numpy as np
#%%
class NLP_LM():
    _order=2 # This is the language model we choose i.e. bigram
    _l_model={} # KneserNeyInterpolated model for vairous ngrams
    _m_model={} # Storing Maximum Likelihood based model for various ngrams
    _vocab=None # Original vocabulary of the input while training
    stemmer = PorterStemmer() # Can be tuned according to situation
    lemmatizer=WordNetLemmatizer() # Since we are using Wordnet basis for language model, this lemmatizer is required.
    word_confidence_level=0.8 # A Tunable parameter, primarily used for spelling match and configuration, but can be tuned.
    max_order=4 # A Tunable parameter indicating the maximum order for the langauge models to be considered. It is tunable.
    tok=["<s>","</s>","<UNK>"] # Indicating start and end of sequences for user inputs
    saved_file_name="lm_model_saves.pkl"
    #
    def __init__(self,corpus=None,order=None,train=False):
        """If being trained on custom data, the data should be a list of texts without tokenization or anything at all. Just a list of texts.
        Do not provide corpus if it is just a running state and not training state. Even if provided, it will be ignored.
        By default if order is specified only that particular order model is initiated and the best model for that order is chosen i.e. an Interpolated language.
        Inputs :
            Corpus : A List of texts, or a pandas series with each text considered as a document in itself. Is required for training.
            train : If the object is initialized as a train_phase or normal use. If normal, pretrained model is loaded.
            order : Pre-chosen langauge model to be chosen. If not given, it tries all the language models upto max_order=4
        """
        self.load=None
        self.model=None
        self._fit_done=False
        self._fixed_order=False
        self.corpus=None
        self.vocab=None
        self.vocab2=None
        #self.train=None
        self._tokens=None # Access by "._texts" method.
        self.org_vocab=None
```

```
        self._train=None
        self._corpus_clean1=None
        self._corpus_clean2=None
        self.avg_scores=[]
        self.vocab_dict=None # Used for word2idx conversions. If any word gets added to corpus it will have to be added here too.
        self.reference={} # Used for reference for readibility and stuff.
        self.context=None
        # This whole thing will go under try.
        # First build empty models if order is given or not depending on it
        if order!=None: #If you want your model to be fixed
            self._order=order
            self.set_models(self._order)
            self._fixed_order=True
            self.choose_model(model_chosen="l_"+str(self._order))
            # We are only going to copy and create class object, in case if we want to operate and run multiple
            # processings i.e. trainings at the same time on the same machine for better testing methodologies.
            #self.model.update({"l_" + str(self._order):self._l_model[self._order].copy()})
            #self.model.update({"m_" + str(self._order):self._m_model[self._order].copy()})
        else:
            for o in range(self.max_order):
                self.set_models(o)
            self.choose_model()
        # If we later want to set the model or change the langauge model, its made available through provided functions.
        # We have to manuall call the set models function if the order is not provided while initialization time.
        # Chief chosen model concept is abandoned, though it can be appropriately activately at relevant places.
        self.given_ng=None # Bigrams of user given text # Needs to be cleared when reset
        # Trigrams of user given text # Needs to be cleared when reset
        self.given_scores={} # Given text Scores # Needs to be cleared when reset
        self.keywords=[] # Keywords in the given user description having the highest score according to differen models is taken here. If no model is specified, all models are clubbed and used.
        self.received=None
        self.given=None
        # A Train method that assimilates all these member functions
        if train:
            if type(corpus)=="NoneType":
                raise ValueError("Corpus is not given or is not in the format specified. Make sure you give it in the proper format as spcified.")
            self.corpus=corpus #Don't forget to give the option to save the corpus once the data processing is done.
            #print(self.corpus)
```

```
            self.start()
            self._build_model()
            print("Building models Finished")
            self._train=None
            self.save()
        else:
            try :
                #obj=NLP_LM.load() # This is wrong, it should turn on model initiation mechanism
                # This needs to be further analyzed. Saved models needs to check for first.
                NLP_LM.load(self)
                #print("----------after loading------------")
                #print(type(self))
                #print(dir(self))
                #print(self._order)
                #print(self._fit_done)
                #print(self.model["l_1"].vocab._len)
                print("Language Models loaded Successfully")
            except:
                raise FileNotFoundError("The pickle file for saved models is not available. Either place it current working directly with name \"lm_model_saves.pkl\".\n Or give the corpus data to train on the ordering given or default ordering.")
    #
    def save(self):
        f=open("./model_saves/"+self.saved_file_name,"wb")
        pkl.dump(self,f)
        f.close()
        print("Language Models are saved")
    #
    @classmethod
    def load(cls,self):
        #print("--------------Before loading----------------")
        f=open("./model_saves/"+cls.saved_file_name,"rb")
        obj=pkl.load(f)
        #print(dir(obj))
        #print(obj._fit_done)
        #print("Total Vocabulary Size loaded :",obj.model["l_1"].vocab._len)
        #print("The length of the Vocabulary is :",len(obj.vocab))
        #print(obj.avg_scores)
        self.__dict__=obj.__dict__.copy() # copying the original data into memory
        f.close()
    #
    @classmethod
    def clean(self):
        """This method is going to clean and delete the objects that are created as part of trianing."""
```

```
        return
    # Signed --- Finished debugging, its working perfect
    def start(self):
        text=self.tokenize()
        #print("Tokenization Done")
        self._tokens=TextCollection(pd.Series(text,dtype="object")) # Access by "._texts" ↵
↳       method.
        self.org_vocab=self._tokens.vocab()
        self._corpus_clean1=self._tokens._texts.apply(self.clean_train) # This is the ↵
↳       primary source for recommendations and similarity
        # After deleting unnecessary words, will have to retain the words
        self._text=TextCollection(self._corpus_clean1)
        self.clean_vocab=self._tokens.vocab()
        #print("Clean Train Done")
        self._corpus_clean2=self._corpus_clean1.apply(self.process)
        #print("Process Training Done")
        self._train,self.vocab=padded_everygram_pipeline(self.max_order,self._corpus_clean2) ↵
↳       #This is hard-coded for safety purposes. Can be changed.
        self.vocab=list(self.vocab)
        self._train=[list(lit) for lit in self._train]
        #self.vocab1=list(set(list(flatten)))
        self.vocab2=list(set(list(flatten(list(self._corpus_clean2)))))
        #self.vocab2=Vocabulary(self.vocab.counts,unk_cutoff=1)
        #print("The Length of the Vocabulary is :",len(self.vocab2))
        self.vocab_dict=dict([(word,key) for key,word in enumerate(self.vocab2)])
        self.create_references()
        self._tokens=TextCollection(self._corpus_clean2)
        #self._tokens=None
        #self._clean_corpus1=None
        #print("Start Up Over")
    # debugging finished.
    #This idea is abandoned, though the method is left alone and should be ignored.
    def choose_model(self,model_chosen=None):
        """Appropriate model is chosen here from the list of available models. If not ↵
↳       provided, all the models in the list will be chosen. However
        it is not recommended for large datasets"""
        if model_chosen:
            try:
                if model_chosen in self._l_model.keys():
                    self.model=self._l_model[model_chosen] # Only copies of the models will ↵
↳                   be taken
                elif model_chosen in self._l_model.keys():
                    self.model=self._m_model[model_chosen]
                # We are also going to completely reset the model and its flavour.
                self.vocab=self.model.vocab
                self.vocab2=Vocabulary(self.vocab.tokens)
```

```
            except:
                self.model=self._m_model[model_chosen]
        else:
            self.model={} # A Dictionary of models will be created.
            # Append all the models to the current models for test case. But, this is not recommended here.
            for each in self._l_model:
                self.model.update({"l_"+str(each):self._l_model[each]})
            for each in self._m_model:
                self.model.update({"m_"+str(each):self._m_model[each]})
            # Can add other models too, if possible.
    #
    # (Keep this). This can be called as many times as required to set the models of various n-grams or further expansions for importing other models from outside
    # Signed -- Verified debugging, working correctly.
    def set_models(self,ordering):
        """ This sets models for whatever order we ask it to and sets it as a class variable. Becomes an object attribute only after fitting"""
        self._m_model.update({str(ordering):MLE(ordering)})
        self._l_model.update({str(ordering):KneserNeyInterpolated(ordering)})
        # can add other language models too, depending upon the situation.
    # change this
    # Signed -- verified debugging, working correctly.
    def _fit_model(self,model_chosen,train_set,vocab):
        """ This fits the models. You can either specify the model name or just pass an empty string in case of fixed order while initializing. """
        train=train_set
        vocab2=vocab
        try:
            if self._fixed_order:
                assert self._fit_done==False
                self.model.fit(train,vocabulary_text=vocab2)
                self._fit_done=True
            else:
                leng=self.model[model_chosen].vocab.counts.keys()
                assert len(leng)==0
                #print(len(leng))
                #print("The vocab size for model",model_chosen,self.model[model_chosen].vocab._len)
                self.model[model_chosen].fit(train,vocabulary_text=vocab2)
                #print(len(leng))
        except:
            print("The language model that is not fitting :",model_chosen)
            raise AssertionError("Fitting already done")

    # Signed -- verified debugging - working properly. Certified
    def _build_model(self):
```

```
        """ Fits the models and builds them up for further analysis.
        NOTE : Currently there is no checking mechanism to see if training for any particular language model is already done.
        So, if new builds or fits are happening, its going to retrain all the models on the given dataset again."""
        # Build a pipeline here that will create a padded pipeline and then fit into each of those models
        if self._fixed_order:
            self._fit_model(model_chosen="",train_set=self._train,vocab=self.vocab2)
        else:
            for each in self.model:
                #print(len(self._train))
                #print(each)
                self._fit_model(model_chosen=each,train_set=self._train,vocab=self.vocab2)
        self._fit_done=True
        #print("fitting Models finished.")
        #Uncomment this after debugging.
        self.avg_scores=self.set_avg_scores()
    #
    # Signed -- Debugging done - Working perfectly.
    def _give_best_words(self):
        """This takes the words already present in the user input prir given, if found any significant words in it based on the score, it will give the words that scored highest. The words with the highest score
        are picked and the input is asked more keep that word in context as in other methods."""
        try:
            self.given_scores!=None
        except:
            raise ValueError("Perhaps validation hasn't been done on atleast one entry yet. Please first try with primary user input.")
        words=[]
        scores=[]
        q=self.given_scores
        for each in q.keys():
            model_param=q[each]
            #print("Model :",each)
            for item in model_param:
                ord_param=model_param[item]
                #print("Order :", item)
                word_score,word_index=ord_param.max(),ord_param.argmax()
                words.append(self.given[word_index])
                scores.append(word_score)
                #print(word_score," : ",word_index)
                #print("Word Suggested : ",my_model.given[word_index])
            #print("----------------------------")
        self.keywords=self.keywords+list(set(words))
```

```
        return words,scores
    #
    #This is a doubtful but not completely functional design. It needs proper checking and support.
    # Signed -- Debugged and Working.
    def recommend_contexts(self):
        """Recommends contexts for the words based on the highest score words that are already given in the description by the user to give him an idea of
        what might be the description. It returns some context words that might help remember or extend the thoughts of the user to input more data."""
        w,s=self._give_best_words()
        #print(w,s)
        self.context=ContextIndex(self._text)
        #print(list(set(self.keywords)))
        kw=[self.reference[wor] for wor in set(self.keywords)]
        #print(kw)
        contexts=[list(self.context.common_contexts([word]).keys()) for word in kw] # Check contexts as a list.
        contexts=[" ".join(word) for each in contexts for word in each]
        return contexts
    # Signed -- Debugged and working.
    def recommend_desc(self): # recommendation texts for users based on the current input
        """Some concordance of top words to given some options for the words and ideas based on the ideas. So, some concordance words are given
        to make the user select statements, or given some descriptions in similar concordance"""
        lines=8
        width=50
        concord=[]
        w,s=self._give_best_words()
        kw=[self.reference[wor] for wor in set(self.keywords)]
        for word in kw:
            recs=self._text.concordance_list(word,width=width,lines=lines)
            for each in recs:
                concord.append(each.line)
        #print(concord[0])
        #for each in concord:
        #    each.left_print
        # Calling function should pick random concordances from these
        #return [print(dir(concs)) for concs in concord ]
        return concord
    #
    #Signed -- Working Correctly
    def recommend_options(self):
        """Finding some common collocations in training set and passing them as option recommendations to improve boosting the description output
        with appending words."""
```

```
        colloc_list=self._text.collocation_list(num=8,window_size=5)
        return [" ".join(each) for each in colloc_list]
    # Signed Debugged -- working as expected.
    def validity_check(self,outside_input):
        """This is all we need for every text input that is given to the chatbot. It only
        handles descriptions over a certain length.
        It doesn't handle anything else.
        The outputs of overall score for different models is given here. It is for analysis
        purpose only. Care should be given that, if any,
        past inputs should be added with the present input to get a full picture.
        Outupt:
        Scores : A Dataframe of Scores by each model on the given chunk of text, given on an
        average
        text : The cleaned corpus text, that is done according the the present module, so
        that it can used further in NN processing.
        """
        #out=[] # The list containing output data for model management engine
        text=outside_input
        text=self.tokenize(text=text)
        try:
            text2=self.clean_test(text)
            #print("Text after clean test",text2)
            text3=self.clean_train(text2)
            #print("Text clean train",text3)
        except:
            print("Exception raised : Not enough text")
            return None
        else:
            text=text3
            text=self.process(text)
            self.received=text
            #print(test)
            text=[self.tok[0]]+text+[self.tok[1]] # Appending end of sentence and start of
            sentence tokens for better analysis
            self.ngram_test(text)
            #Here model is not yet decided along with proper model mechanism
            scores={} # This obtained score is the score for ngram for the given model. so,
            here model represents the rows not columns
            for each_model in self.model:
                model=self.model[each_model]
                n=2 #staring with bigrams for contexts, though unigram can be used for
                individual probability scores.
                obtained_scores={}
                for each_ng in self.given_ng:
                    #print("The lengh of the ngram is :",len(each_ng))
                    current_ord=len(each_ng[0])
                    #print(current_ord)
```

```
                    # Each is a bigram, trigram and quadgram
                    #obtained_score.append(self.ngram_score_calculator(each_ng,n,model))
                    s=self.ngram_score_calculator(each_ng,n,model)
                    obtained_scores.update({current_ord:np.array(s)})
                    scores.update({each_model+"_order_"+str(n):sum(s)})
                    n+=1
                if n==self.max_order:
                    n=2
                #print(type(obtained_scores[0]))
                self.given_scores.update({each_model:obtained_scores})
                #scores.update({each_model:obtained_score})
            #s=[self.ngram_score_calculator(test,i+2,model) for entry  for i in ↵
↳           range(len(test))]
            #score_bigram_l_model=self.ngram_score_calculator(self.given_bi,2,self.l_model_2)
            #score_trigram_l_model=self.ngram_score_calculator(self.given_tri,3,self.l_model_3)
            #score_bigram_m_model=self.ngram_score_calculator(self.given_bi,2,self.m_model_2)
            #score_trigram_m_model=self.ngram_score_calculator(self.given_tri,3,self.m_model_3)
            #self.given_scores.append(score_bigram_l_model)
            #self.given_scores.append(score_trigram_l_model)
            #self.given_scores.append(score_bigram_m_model)
            #self.given_scores.append(score_trigram_m_model)
            text.remove(self.tok[0]) # Removing start of sequence token
            text.remove(self.tok[1]) # Removing end of sequence token
            # Add 2 more different scores
            self.given=text
            ↵
↳           #out=out+[sum(score_bigram_l_model),sum(score_trigram_l_model),sum(score_bigram_m_↵
↳           model),sum(score_trigram_m_model),text]
            #print(test)
            #print("Validity of the input is checked")
            return [pd.Series(scores),self.given]#test#self.given#out
    # Signed -- Working Correctly.
    def ngram_test(self,text_in):
        """Takes in text tokens and generates a dataframe of all kinds of ngrams added to it ↵
↳       and gives back the dataframe"""
        #temp=pd.Series(text_in)
        #temp=pd.DataFrame(text_in,columns=["tok"])
        self.given_ng=[list(ngrams(text_in,n=order)) for order in range(2,self.max_order+1)]

    # Signed and approved -- Debugging done.---  Perfectly working.
    def set_avg_scores(self):
        """This calculates the average scores for given training set that is used for ↵
↳       comparison.
        This is the heart of the language model with score setting and all. So, its crucial"""
        temp=pd.DataFrame(self._corpus_clean2,columns=["tok"])
        #max_order=5
        models_peep=[]
```

```
        i=0
        for model_given in self.model:
            try:
                i+=1
                model=self.model[model_given]
                scores=[]
                #print("-----------------------")
                #print("model :",model_given)
                for order in range(2,self.max_order+1):
                    #temp["ngrm_"+str(order)]=temp["tok"].apply(ngrams,n=order).apply(list)
                    k=temp["tok"].apply(ngrams,n=order).apply(list)
                    #print("Order :",order)
                    k=k.apply(self.ngram_score_calculator,n=order,model=model).apply(sum)
                    #k = k.rename(columns={"tok":order})
                    k.name=model_given+"_order_"+str(order)
                    #print(k.name)
                    scores.append(k)
                models_peep.append(pd.DataFrame(scores).T)
                if i==self.max_order:
                    i=0
            except:
                print("Not working for the model :",model_given)
        ret=pd.concat(models_peep,axis=1) # This is the complete dataframe created for further analysis and is the important part of language models.
        print("Langauge Model Average Scores are Set")
        return ret
    #Debugged -- Working perfectly.
    def ngram_score_calculator(self,ngram_list, n, model):
        """This create ngram model's score on a given corpus's ngram list for a given entry. The choice is ours. It will return the scores for each and every
        word in the list of words."""
        score=[]
        for each in ngram_list:
            try:
                score.append(model.score(each[0],[each[i] for i in range(1,n)]))
            except:
                print("Not working out",each)
        #print(score)
        #print("-----------------\n")
        return score
        #entry_score=[]
        #s=[]
        #for each in ngram_list:
        #    print(each)
        #    s.append(model.score(each[0],[each[i] for i in range(1,NLP_LM.max_order)]))
        #entry_score=[model.score() for each in ngram_list]
        #print(entry_score)
```

```
        #return sum(entry_score)
    #
    def clean_test(self,tokenized_text):
        """This will handle all the unknown words that are given by the user but not in the train corpus. If not present then it will return the
        will check for spelling mistakes and if any will correct it based on certain threshold confidence level. If everything is okay, then synonyms of the word
        that are present in trained corpus will be found out and replaced appropriately."""
        text=tokenized_text
        #spell=SpellChecker()
        to_remove=[]
        for index in range(len(text)):
            each_word=text[index]
            #print(each_word)
            # Have to carefully checkout if the corpus and vocab of the set is the same.
            #print(each_word in self.org_vocab)
            if each_word in self.org_vocab:
                continue
            else :
                #print("Word not in vocabulary :",each_word)
                #First check if its in english words or not.
                if each_word in eng_words.words():
                    # Addressing the presence of new word in the description given
                    synonyms=[]
                    for synset in eng_words.synsets(each_word):
                        # creating a synonym set here
                       for lemma in synset.lemmas():
                           synonyms.append(lemma.name())    # Creating list of Synonyms
                    #print(synonyms)
                    syns=copy.copy(synonyms)
                    try:
                        assert len(synonyms)!=0
                        #syns=synonyms
                        for word in synonyms:
                            #print(word)
                            # Finding if the word is in the original vocabulary
                            if word in self.org_vocab:
                                #print("Got the Word ",word)
                                text[index]=word
                                break
                            else:
                                syns.remove(word)
                                #print(syns)
                            #print(len(syns))
                    except:
                        #print("Exception Reached")
                        #simply delete the word from the description and consider it as a
```

```
                        foriegn language word or something
                        to_remove.append(text[index])
                    else:
                        if len(syns)==0:
                            to_remove.append(text[index])
                else:
                    # It is assumed that all spelling correct words are already included in the wordnet.
                    words = suggest("thigns").spellcheck()
                    for word,confidence in words:
                        if confidence>self.word_confidence_level and word in self.org_vocab:
                            #Addressing spelling mistake
                            text[index]=word
                            break
        #print("Testing Clean Done")
        #print ("After clean",text)
        #print ("To delete words",to_remove)
        for word in to_remove:
            text.remove(word)
        #print("AFter cleaning",text)
        return text
    # Signed -- working properly.
    def generate_text(self,*args,**kwargs):
        """You have to supply the model you want to use from among the available models, specify the number of words by n=number_of_words, specify the random seed by giving kwargs random_seed=value, specify the text_seed=["list","of","words"] kind of methodology.
        Returns a list of keywords"""
        out=[]
        for model_name in self.model:
            model=self.model[model_name]
            out.append(model.generate(*args,**kwargs))
        return out
    ###########################################################################
    # Do NOT Touch this section. It is static.
    ###########################################################################
    #
    #Keep this
    # Signed and approved --  working after debug (perfect)
    def tokenize(self,text=None):
        """This is a simple tokenizer for data analysis and generating tokens."""
        k=[]
        if type(text)==str:
            k=word_tokenize(text)
        else:
            try:
                for each in self.corpus:
```

```
                    #print("----------------------")
                    #print(each)
                    k.append(word_tokenize(each))
            except:
                raise TypeError("Given value is not a String type")
        #print(k)
        #print("Tokenization Done")
        return k
    #
    # Signed and approved -- working after debug
    def process(self,text_token_list):
        """lemmatize and stem the words"""
        text=text_token_list
        text=[self.lemmatizer.lemmatize(word) for word in text]
        text=[self.stemmer.stem(word) for word in text]
        #print("Processing Done")
        return text
    #
    # Signed and approved -- working after debug
    def clean_train(self,texts_token_list):
        """This will clean the data with elimination of punctuations and non-english words, ↵
↳       and stop words"""
        text=texts_token_list
        text=[word for word in text if word not in punct]
        text=[word for word in text if not word.isdigit()]
        text=[word for word in text if word in eng_words.words()]
        stopwords = sw.words('english')
        text=[word.lower() for word in text if word not in stopwords]
        #print("Training Clean Done")
        return text
    #
    #
    def create_references(self):
        """This creates references for the dataset, so that information can be properly ↵
↳       conveyed and
        sent to the user. This is used so that stemming can be reversed to make the data ↵
↳       more readable."""
        org_data=self.clean_vocab
        stem_data=self.process(org_data)
        self.reference=dict(zip(stem_data,org_data))
    #
    ##########################################################################
    #
    ##########################################################################
```

d:\Work_folders\code\python-Spyder\Shravani-Chatbot\language_models.py

#%%

d:\Work_folders\code\python-Spyder\Shravani-Chatbot\model_management_engine.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Sep 22 15:50:27 2021
This module will define a class of model management engine that is going to bring three different model managers as parts into
a single engine that will interface with the chabot.

NOTE : There should be a better automatic threshold set for prediction based on the average score of equal likelihood among the given classes. So, here it is directly
coded to have _threshold_confidence_AL as 0.40. Since there are five accident levels, random probability will be 20 for each. So, double of any probability eliminates (very crudely)
any chance of likelihood of it being a random event or chance prediction. So, threshold is set to 40% or 0.40. It can be tuned that way, but way more analysis is to be done before taking
that decision, especially conservative estimations. So, plan ahead for that.

@author: Sheshank_Joshi
"""
#%%
import language_models
import NLP_core_manager
import supervised_core_manager
import numpy as np
import pandas as pd
import json
import copy
from scipy import stats
import pickle as pkl
import model_training_engine as me
#%%
class model_management_engine():
    """This is the original management engine that brings in 3 different managers together. The methods are controls lie here for thresholds. Prelimnary integrity check also happens here
    for proper finding of the files. Since the corresponding managers are responsible for checking file integrities and saved models, it is delegated to the corresponding managers."""
    _threshold_confidence_AL=0.40 # This means any given prediction from the model is having atleast one prediction that has 45% accuracy in any one prediction i.e. just above average.
    _prob_threshold_score=0.25
    parameter=1 # This paramter decides how strict our predictions or uncertainity we can allow in our NN model.
    #
    def __init__(self):
        self._lm=None
        self._NN=None
```

```
        self._sup=None
        self.vocab_dict=None # This is the master vocabulary used for word indexing both for testing and training.
        # More parameters that actually control the mechanism is given here.
        self.sample_scores=None
        self.AL_prediction=None
        self.tokens=None
        self.models_imp=["l_2_order_2",'m_2_order_2'] #,'m_3_order_3','l_3_order_3']
        # As is mentioned, only second order bigram data is considered
        self.init_lm()
        self.init_NN()
        self.init_sup()
        self.options=self.load_options_data()
        self.enc=None
        self.encode()
        print("Management Engine Initializing done")
    #
    def init_lm(self):
        try:
            self._lm=language_models.NLP_LM()
            print("Language model is initalized")
            #print(len(self._lm.vocab_dict))
        except:
            # Have to change this based upon the
            raise FileNotFoundError("The language_models module doesn't exist. Please place it in the same folder as this file")
    #
    def init_NN(self):
        #Have to write this so that appropriate model is chosen here itself.
        try:
            self._NN=NLP_core_manager.NN_NLP()
            print("Neural Network Model is Initialized")

        except:
            raise FileNotFoundError("The NLP_core_manage module doesn't exist or is tampered with. Please place it in the same folder as this file")
    #
    def init_sup(self):
        try:
            #print("working on")
            self._sup=supervised_core_manager.sup_manager()
            print("Supervise Learning Model is Initialized")
        except:
            raise FileNotFoundError("The Supervised models are not found")
    #
    # Signed - Debuggig done -- Working Perfectly.
    def predict_AL(self,prob=False):
```

```
        """This will handle all the prections related to accident level.Ideally should use
        more than one model to do the actual prediction and choose the average and best
        ranking among the models.
        But here only one particular model is used along with analysis of all the given four
        prediction.
        Basically, the four probability outputs given out are taken and checked for the
        cumulative differences. Then a skew data is
        framed across all the predictions. If that skew threshold is crossed by how many
        argmax and the maximum vote on that
        is taken as the final output.
        NOTE : That is not implemented in this as of now. Instead it is touched upon, just
        how many standard deviations a particular argmax is above the rest. If it
        is significant, then we consider that output. Else we take the conservative estimate.
        If there are two predictions, then the highest prediction is should be atleast one
        std away from the one next in its line.
        Returns : The most conservative estimate if the probability of the prediction is
        nearly similar to any two given cases."""
        if prob:
            return self.AL_prediction
        else:
            prediction=self.multimodal_predict()
            #print(prediction)
            votes=[]
            for each in prediction:
                votes.append(each.argmax())
            votes=pd.Series(votes)
            #print(votes)
            c=pd.Series(votes).value_counts()
            #print(c)
            i=0
            try:
                while 1:
                    #print(i)
                    #print(c[i])
                    #print(c[i+1])
                    if c[i]==c[i+1]:
                        i+=1
                        continue
                    else:
                        print(i+1)
                        break
            except:
                #print("The Length of the series is :",len(c))
                pass
            final_prediction=c.index[i]
            #print("Final Prediction is :",final_prediction)
            return final_prediction
```

```
            #print(c)
            #print (prediction)
            #return prediction
    #
    # Signed - Debugging Done -- Working perfectly.
    def multimodal_predict(self):
        """This function gives prediction from multiple models."""
        #toks=self.word2idx(self.tokens)
        #print(toks)
        #X_test=self._NN.pad_sequences([toks])[0]
        #print(X_test)
        #models=None # These are to be delegated to the outside module
        #predictions=[]
        #for each in models:
        #    predictions.append(models.predict(X_test))
        predictions=self._NN.predict(self.tokens)
        #print(self.tokens)
        return predictions
    #
    def encode(self):
        """Encode the test data into the same mechanism and working as that of the original ↵
↳       training data."""
        file=open("./model_saves/encoder.pkl","rb")
        enc=pkl.load(file)
        self.enc=enc
    #
    def predict_sup(self,entry):
        """takes in entire dataframe (along with accident level) and then tries to predict ↵
↳       the potential accident level."""
        #Delegate to supervised model manager.
        # make a one hot encoder on the train data and then fix it up with test data.
        df=pd.DataFrame(entry,index=[0])
        #print(df)
        df=self.enc.transform(df).toarray()
        #print(df)
        predictions=self._sup.predict(df)
        # use the predictions given by various models here just like predict_AL funciton and ↵
↳       then use them.
        k=pd.Series(predictions)
        c=pd.Series(k).value_counts()
        #print(c)
        i=0
        try:
            while 1:
                #print(i)
                #print(c[i])
                #print(c[i+1])
```

```
                if c[i]==c[i+1]:
                    i+=1
                    continue
                else:
                    #print(i+1)
                    break
        except:
            #print("The Length of the series is :",len(c))
            pass
        final_prediction=c.index[i]
        #print("final prediction :",final_prediction)
        return final_prediction
    #
    # Signed -- debugging Done working perfectly.
    def predict_rough(self,corps):
        """This handles a rough accident level prediction to go with go ahead mechanism that will return a Go ahead or not as a boolean"""
        word_nums=self.word2idx(corps)
        word_nums=self._NN.pad_sequences([word_nums])
        #print(word_nums)
        best_model=self._NN.model_best
        #print(best_model)
        probs=self._NN._models_list[best_model].predict(word_nums) # This is a single model prediction, not a multi-model prediction. So, can't be finalized.
        # Checking if the highest probability of any given prediction is greater than threshold
        if probs.max()>self._prob_threshold_score:
            self.AL_prediction=probs.argmax()
            self.tokens=word_nums
            return True
        else:
            return False
    #
    # Signed - Debugging Done -- working perfectly.
    def generate_options(self,option):
        """Depending upon the option given there will be suggestions by the function that will be used to give user suggestions or thoughtful idea to the user.
        It will always give four options in random."""
        if option==1:
            choices=self._lm.recommend_desc()
        elif option==2:
            choices=self._lm.recommend_contexts()
        elif option==3:
            choices=self._lm.recommend_options()
        #choices=list(set(choices))
        indices=np.random.randint(0,len(choices),5)
        ret=[choices[opt] for opt in set(indices)]
        return ret
```

```
    #
    #
    def avg_score(self):
        """Will fetch a bunch of scores from downstream and returns the threshold best score. We can check here for more detailed one, even though it is not
        implemented here extensively."""
        # Here there should be code for analyzing the average scores and choosing the best language model
        # and which one is the best one to better decision making. But, here l_model trigram is used for testing purposes.
        scores=self._lm.avg_scores.describe().loc["25%"]
        scores=scores[self.models_imp].mean()
        # Within 15% of that tolerated mean value score is acceptable, though the minimum value is what is required. But here
        # 25% mean average value is taken as a threshold value.
        return scores
    #
    # Signed - Debugging Done -- Working Perfectly.
    def fetch_score(self,text_in):
        """Will pull a bunch of scores, return the score,and the corresponding corpus of the data"""
        options=self._lm.validity_check(text_in)
        if options:
            # Has options i.e. tokenized texts and a bunch of other scores to
            # Code for analyzing the fetched scores more deeply after seeing the working conditions and the logic for all these.
            # But for now, implementing only l_model trigram score. Similary, only
            temp=options[0]
            self.tokens=options[1]
            temp=temp[self.models_imp].mean()
        else:
            # This case came up means there is not a single useful keyword in the description
            temp=0
            self.tokens=[]
        return temp,self.tokens
    #
    # Signed - Debugging Done -- Working Perfectly.
    def word2idx(self,text_in):
        word_dict=self._lm.vocab_dict
        return [word_dict[tok] for tok in text_in]
    #
    # Signed - Debugging Done -- Working Perfectly.
    def load_options_data(self):
        """This is going to load the options data that was created as part of training.Yet to be worked upon."""
        try:
            file=open("./model_config/options_data.json","r")
```

```
            data=json.load(file)
            file.close()
        except:
            raise FileNotFoundError("The Trained file is not to be found")
        else:
            #_=data.pop("Accident Level")
            return data
    #
```

```
# -*- coding: utf-8 -*-
"""
Created on Fri Sep 10 17:13:28 2021
This is the generalized NLP interface that handles both training and the language models
left and right to the current situation.

NOTE to self : Write a Function that will choose the best performing model from the available models, if required.
@author: Sheshank_Joshi
"""
#%%
import numpy as np
import pandas as pd
import tensorflow.keras as k
import NLP_core_manager
import language_models as lm
import supervised_core_manager
import json
from imblearn.over_sampling import RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler
import pickle as pkl
from sklearn.preprocessing import OneHotEncoder
#%%
class models_training_engine():
    """The Interfacing class that acts a bridge between Language model, Neural Network model and supervised model for training purposes. Automatic training
    doesn't happen here. We have to manually step by step procedures as listed below
    1. set description column name
    2. set dependent column name
    3. set target column name
    4. Call the Train function (this will take care of the rest in the background)"""
    #_max_len=45 # Tunable parameter to the maximum length of the description for the words
    _lm=None # Where we store the Language models Manager
    _NN=None # Where we store the Neural Network Manager
    _sup=None # Where we store the supervised Model Manager
    def __init__(self,dataframe):
        """It is user's responsibility to pass pandas dataframe object to the specifications as given in the dataset."""
        self.df=None
        self._check_dataframe(dataframe=dataframe)
        self._target_col=None # The Original Target Col name i.e. The Potential Accident Level in our case
        self._desc_col=None # The name of the Description column, that contains the description of the incident.
        self._desc=None # Actual description column that has text data i..e non tokenized, original corpus with individual texts
        self._target=None # The pandas series that contains the potential accident level.
```

```
        self._data=None
        self.dep_col=None
        self.options_data=None
        self._target2=None
        self.enc=None

    #Signed -- Debugging -- Working correctly. Rank 0
    def _check_dataframe(self,dataframe):
        """Just trying to catch an exception in case the supplied data isn't a pandas dataframe."""
        try:
            assert type(dataframe) == type(pd.DataFrame())
            self.df=dataframe
            try:
                self.df.drop("Data",axis=1,inplace=True)
            except:
                pass
            try:
                self.df.drop("Unnamed: 0",axis=1,inplace=True)
            except:
                pass
            #print(self.df.columns)
            #self._data=dataframe
        except:
            raise TypeError("The Object passed is not a Pandas Dataframe object. Please check it and re-initialize")
    #
    # Signed - Debugging -- Working Correctly. Rank 1
    def set_desc_column(self,name):
        """Setting the column name for Description"""
        if self._desc_col:
            print("The Description column is already set.")
        else:
            #print(name)
            try:
                name in self.df.columns
                self._desc_col=name
                self._desc=pd.DataFrame(self.df.pop(name))
                #self._desc["orig_length"]=self._desc[name].apply(len)
                # This can be completely avoided.
            except AssertionError:
                print("There is no target_column in the dataframe. Please change dataframe or please change the target name")
            else:
                pass
                #print("Please also set target column for Potential Accident Level")
    #
```

```
    # Signed - Debugged -- Working Correctly. Rank 2
    def set_dep_target(self,name):
        """Set the dependent element target here. It needs appropriate labeling."""
        if not self.dep_col:
            self.dep_col=name
            self._desc[self.dep_col]=self.df[self.dep_col]
        else:
            print("Dependent column is already set")
    #
    # Setting Target column for Supervised Model
    # Signed - Debuggin -- working correctly. Rank 3
    def set_target_column(self,name):
        """Sets the Target Column for final prediction"""
        #convert=self.options_data
        if not self._desc_col:
            raise ValueError("First Set the Description Column by calling appropriate method")
        elif self._desc_col==name:
            raise NameError("Please select other column for Supervised model Target column. ↵
↳           Given column {}".format(self._target))
        else:
            try:
                assert name in self.df.columns
                self._target_col=name
                ↵
↳               self._data,self._target=self.shape_resampling(self.df.drop(name,axis=1),self.d↵
↳               f[name])
                self._target2=self.df.pop(self._target_col)
                #convert=dict([(value,key) for key,value in ↵
↳               enumerate(convert[self._target_col])])
                #self.target2=self._target.replace(to_replace=convert)
            except:
                raise ValueError("Sorry, the value you have in input is not in the columns")
    #
    # Signed - Debugging Finished -- Working Correctly. Rank 5
    def _lm_initialize(self):
        """This will initialize the Language Model, and set things up for other trainings to ↵
↳       happen. This is the crucial
        step to make any changes for any further analysis."""
        try:
            if not self._desc.empty:
                # order is not going to be specified here.
                self._lm=lm.NLP_LM(corpus=self._desc[self._desc_col],train=True)
                print("Language Model Trained Successfully")
            else:
                raise ValueError("The Description column is not set yet. Check about it.")
        except:
            #self._desc=None
```

```
            print("There is something wrong with the Description given")
            raise AttributeError("Description Column not appropriate")
    #
    # Signed - Debugging Done -- Working Correclty. Rank 6
    def _NN_initialize(self):
        """This will initialize the Neural Network Model and set things up ready, including saving the models."""
        try:
            # assert self._lm
            # assert self.dep_col # Checking if the dependent column is set or not.
            # Here vocab needs to be checked if it is appropriate. We will build all our models initially
            data_corpus=self._lm._corpus_clean2
            data_corpus=data_corpus.apply(self.word2idx)
            self._desc[self._desc_col]=data_corpus

            self._NN=NLP_core_manager.NN_NLP(dat=self._desc[[self._desc_col,self.dep_col]],targ=self.dep_col,vocab=self._lm.vocab,train=True,auto=True)
            print("Neural Networks Trained")
        except:
            raise ValueError("You need to first specify what is the text, dependent column.")
    #
    #Signed - Debugging Done -- Working Perfectly. Rank 7
    def _sup_initialize(self):
        """This will initialize the Supervised Learning Model and will set things up, including saving the models for later."""
        try:
            self._create_data_dictionary()
            #print(self._data.columns,self._target.name)
            x,y=self._data,self._target
            #print(y)
            #print(self.options_data["Accident Level"])
            #convert=dict([(value,key) for key,value in enumerate(self.options_data[self.dep_col])])
            #y=y.replace(to_replace=convert)
            #print(x.shape,y.shape)
            #print(x.columns)
            self._sup=supervised_core_manager.sup_manager(X=x,y=y,train=True,auto=True)
            print("Supervised Model Trained.")
        except:
            raise ValueError("There is something wrong with the supervised model. Check if files are available")
    #
    # Signed - Debugging Done -- Working Correctly. Rank 4
    def _create_data_dictionary(self):
        """Will create a dictionary for values and encoding appropriately in the order in which they will be fed to supervised
```

```
        learning model. The Date column is completely ignored here. Further thoughts about including it as a time series is to be
        seen much later."""
        #Remove Get
        df=self._data
        encoder=OneHotEncoder(handle_unknown='ignore')
        # Should be called in by the supervised model trainer. It should take care of the whole thing.
        cols={}
        #print(self.df.columns)
        try:
            df=df.drop(["Data"],axis=1)
        except:
            pass
        try:
            df=df.drop(["Unnamed: 0"],axis=1)
        except:
            pass
        #print(df.columns)
        encoder.fit(df)
        x=encoder.transform(df).toarray()
        #print(x)
        for index in range(len(df.columns)):
            col=df.columns[index]
            cols.update({col:list(encoder.categories_[index])})
        #temp=pd.get_dummies(self._target)
        #self._target=temp
        #y=encoder.transform(_target)
        #cols.update({self._target_col:list(temp.columns)})
        #
        self.options_data=cols
        file=open("./model_saves/options_data.json","w")
        #json_obj=json.dumps(col)
        json.dump(self.options_data,file)
        file.close()
        #
        f=open("./model_saves/encoder.pkl","wb")
        #json.dump(encoder)
        pkl.dump(encoder,f)
        f.close()
        #
        x=pd.DataFrame(x,columns=[item for cat in encoder.categories_ for item in cat])
        self._data=x
        #print(self._data)
    #
    # singed -- Debugged -- working correctly.
    def word2idx(self,text_in):
```

```
        """Used to convert the words into appropriate indexed words."""
        word_dict=self._lm.vocab_dict
        return [word_dict[tok] for tok in text_in]
    #
    # This is not at all needed.. Keep it aside.
    def train(self):
        """This function is going to train all the NN Models followed by all the Supervised
        models and then place them in particular
        order in their machine in appropriate order for a standby call."""
        self._lm_initialize()
        self._NN_initialize() # We will make extensive use of multiprocessing methodologies.
        self._sup_initialize()
        #self.optimize() # Decision on whether one single model should be used is not yet
        decided.
    #
    #
    def optimize(self):
        """This function is going to call all the models invovled and optimize them
        appropriately within the specified parameters given in
        their respective Model managers"""
        # Code calling for optimization in both supervised learning model and NN Training
        model for chosen models.
        return
    #
    #
    # Signed - Debugging done -- Working perfectly.
    def shape_resampling(self,X_train,y_train):
        """This will try to eliminate any Class imbalances observed within the data."""
        ros = RandomOverSampler()
        rus = RandomUnderSampler()
        X_train,y_train=ros.fit_resample(X_train,y_train)
        X_train,y_train=rus.fit_resample(X_train,y_train)
        return X_train,y_train
    #


#%%
```

d:\Work_folders\code\python-Spyder\Shravani-Chatbot\NLP_core_manager.py

```
# -*- coding: utf-8 -*-
"""
Created on Thu Sep  9 20:10:06 2021
This is the left hand module that contains the core interfacing for NN Models. The models are placed in a separate file. Only example models are placed in this. So, modularity
and scalability is achieved.

NOTE : A Class decorator function is required here that will pass all the methods to all the models stored in the manager at the same time and will work in exact class methods.

NOTE to Self : Try to implement the classifier models for proper guess and evaluating the self model based on how good our model is going to predict things, based on
validation input given the user by selecting the country etc.

NOTE to self : This needs some model management to be done. Saving the parameters and then serving them appropriately. Can use pickling the model and then resue them for prediction, along with
all the other notes. It should be quite visible.

@author: Sheshank_Joshi
"""

#%%
import tensorflow as tf
from tensorflow import keras as k
import NN_models as NN
import pandas as pd
import numpy as np
#from tensorflow.keras import models
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import CategoricalCrossentropy
from tensorflow.keras.metrics import CategoricalAccuracy
from tensorflow.keras.models import load_model
from pathlib import Path
import pickle as pkl
import json
import copy
#X=np.random.random((50,12))
#y=np.random.randint(0,2,(50,5))

#%%
class NN_NLP():
    """ The Class contains a bunch of models that is shown through its structures by calling the method  "show_models". The actual creation of models
    is given a choice at the time of initiation of the model object and then trained
```

```
    appropriately. All models are not pre-trained, though it can be done and called
    upon purpose and situation. We can choose to see the model architecture from "show_model_arch"

    Parameters chosen :
    optimizer_chosen : Intance of optimizers from Tensorflow Library or custom Optimizers
    losses_chosen : Losses of custom function or from Tensorflow Library
    metric_chosn : Metrics from a custom function or from Tensorflow Library

    NOTE : If appropriate model is not chosen, the Tensorflow functions and customizations will not be available. A single model
    needs to be chosen to increase the customization options and fine tuning the model.

    """
    # These are the default parameters to be used on all the models.
    max_len=50 # Maximum length of the sequences
    Embedding_dimensions=50 # Tunable parameter for better success.
    units=48
    #saved_file_name="NN_NLP.pkl" # This idea is not working, tested properly.
    epoch=55
    ######################################################
    # Initialize Function.
    ######################################################
    def __init__(self,dat=None,targ=None,vocab=None,train=False,model_chosen=None,auto=False):
        """Initializing is delegated to initiate method so multiple models can be initialized but only one running at any given time.
        Data : Should be index converted words i.e. numbers not words. Padding is not needed. Should have Target prediction along with Text Columns.
        vocab : Passing vocabulary size is enough, though passing entire vocabulary is also accepted.
        target : Should specify the name of the target column within the dataframe given
        model_chosen : specify the name of the model you want to initiate with. If none is given all the models will be initiated (waste of resources).
        Can be later chosen, if not sure what are the models avaiable, just use the method NN_NLP.show_available_models() and then choose from the given strings.
        auto : If you want automatic fitting and compiling with default best-chosen parameters for the models or not. If not, be careful to save your models using obj.save()
        method."""
        self.vocab_size=None # Stores the list vocabulary of the original data (imported from Langauge Model)
        self.data=None # Stores the Original data without the target column and exclusively the text corpus as a Series
        self.data2=None
        self.target=None # Stores the target data in a Pandas dataframe, with dummies stored in it in floating point.
        # self.model=None # This is the place one big chosen model resides
```

```
        self._models_list={} # This is the place where all the models in the NN_models are ↵
↳       imported and stored as standby with best default parameters.
        # Once the models are initiated or chosen, access to the parameters for tuning the ↵
↳       model become available like all the variables within the model
        # which are listed by dir.
        self.models_built=False # A Check whether the models are built or not
        self._model=None # If a specific model is initiated it is loaded here else it will ↵
↳       remain None.
        self._initiated=False # Check the models have been initiated or not, especially when ↵
↳       training purposes
        # These are the parameters chosen after extensive testing based on specifications ↵
↳       made on the training set.
        self.optimizer_chosen="adam"#Adam(learning_rate=0.01) # Setting the default ↵
↳       optimizer chosen
        ↵
↳       self.loss_chosen=CategoricalCrossentropy()#"categorical_crossentropy"#CategoricalCross↵
↳       entropy() # setting the default loss; can be reset anytime and models rebuilt.
        self.metric_chosen=CategoricalAccuracy()#"categorical_accuracy" ↵
↳       #CategoricalAccuracy() # setting the default metric; can be reset anytime and models ↵
↳       rebuilt.
        self._fit_done=False
        self.model_compiled=False
        self.model_best=None
        self._AL=None
        self.t=None
        self.call_back=EarlyStopping(monitor='categorical_accuracy',
                                     min_delta=0.05,
                                     patience=10,
                                     verbose=1,
                                     mode='auto',
                                     baseline=0.90,
                                     restore_best_weights=True)

        #
        if train:
            # This only prepares the data given for training. You call need to call train ↵
↳           method to train chosen model or bunch of models.
            data=dat
            target=targ
            #print(data)
            #print(target)
            try:
                data.empty
                #print(type(data))
                #print(data.empty)
                #assert data.empty==True
                #print(vocab)
```

```
                #assert data==None
                try:
                    vocab!=0
                except:
                    try:
                        len(vocab)==0
                    except:
                        raise ValueError("The Vocabulary length can't be zero")
                #assert target.empty
                type(data)==type(pd.DataFrame())
                assert target in data.columns
            except:
                raise ValueError("The Arguments for Vocab and Data is not provided right. Please provide them")
            else:
                if (type(vocab)==int) | (type(vocab)==float):
                    self.vocab_size=vocab
                else:
                    self.vocab_size=len(vocab)
                self.data=data # Setting up the data.
                self.target=target # Setting up the Target
                self.setup(train) # setting up the data and target
                if model_chosen:
                    self.initiate(model_chosen) # Directly initiate the model that is chosen with no hassles.
                else:
                    self.initiate() # Initiate all the models possible with default paramters, best chosen. Can be retrained appropraitely through objects after accessing or selecting.
                self.build_models()
                 # Will show model names and encourage to choose from
                # Now We have to introduce a method to train all the models or the chosen model with input data
                if auto:
                    #print("entered auto")
                    #self.setup(train)
                    #self.compile_model() # Automatically Fits the data
                    self.fit(batch_size=8,epochs=self.epoch,use_multiprocessing=True) # Automatically Fits the data
                    self.save_models()
                    self.model_best,_=self.choose_best_model()
                else:
                    self.show_models_names()
        else:
            try:
                #NN_NLP.load(self) # Loading the self object with data.
                self.load_models()
```

```
                print("Neural Network Models are loaded Successfully")
            except:
                print("There is something wrong here")
                raise FileNotFoundError("Model Files aren't found. Please check it.")
            # Write code to restore the data into the models, includeing the object state ↵
↳           and also the model weights and load them appropriately.
        #self._classifier=None   # This is not being dealt with currently.
        ## Write script to save the entire object as it is given here, and then after ↵
↳       loading object, appropriate
        # model weights are to be loaded accordingly, when the entire object is pickled.

    ######################################################
    #  EOF
    ######################################################
    # Signed Debugging -- Working Correctly.
    def setup(self,train):
        """This sets up the models and the entire management core if training is chosen. ↵
↳       Just an alias procedure."""
        if train:
            self.data=self.data.sample(frac=1).reset_index(drop=True)
            self.t=copy.copy(self.target) # Saving the Target Variable column name.
            self.target=self.data.pop(self.target) # This is the y for training data
            self.target=self.prepare_target() # Preparing the actual target data by ↵
↳           generating the
            self.data2=self.data[self.data.columns[0]]
            self.data2=self.pad_sequences(self.data2) # Padding the sequences with default ↵
↳           paramters for training directly.
             # All the models are built with a default input shape, but none of the models ↵
↳            are actually used, until specified.

    #
    #
    def choose_best_model(self):
        """This automatically sets the best model based on the situation and the ↵
↳       characteristics. The Criteria is
        median accuracy. As the accuracy can change, median value is a good indicator of how ↵
↳       stable the model is. As the
        accuracy keeps increasing the median value keeps shifting, or it will say the same.
        This only works if there is no chosen model. """
        model_character={}
        information=[]
        #print("\n-------------------------------------------------\n")
        for each in self._models_list:
            try:
                model=self._models_list[each].history.history
                #print(model)
                #print(each)
```

```
                data_info=pd.DataFrame(model,columns=list(model.keys())).describe()
                model_character.update({each:data_info})
                #print("The Mean Loss is :",data_info["loss"].loc["min"])
                #print("The Maximum Accuracy is ↵
↳               :",data_info["categorical_accuracy"].loc["max"])
                #print("The Median Accuracy is :",data_info["categorical_accuracy"].loc["50%"])
                #print("----------------------------------")
                median=data_info["categorical_accuracy"].loc["50%"]
                #print("The Median value is :",median)
                information.append(median)
            except:
                print("Choosing Best Model is not working out for the Model :",each)
        ind=information.index(max(information))
        model_chosen=list(model_character.keys())[ind]
        return model_chosen,model_character
    #
    #Signed - Debugging done -- Perfectly Working
    def save_models(self):
        """This is going to save the models exactly as they are. They can be retrained on ↵
↳       additional data."""
        for each in self._models_list:
            model=self._models_list[each]
            try:
                model.save("./model_saves/"+model.name)
            except:
                print("Couldn't save Model :",each)

        try:
            file=open("./model_config/NLP_data.json","w")
            AL_data=dict([(key,value) for key,value in enumerate(self.target.columns)])
            data_to_store={self.t:AL_data,"best_model":self.model_best}
            json.dump(data_to_store,file)
            file.close()
        except:
            print("There seems to be some problem with saving data. Will not save them. But ↵
↳           beware")

    # Signed - Debugging done -- Perfectly Working
    def load_models(self):
        """Models and their weights, both of them are saved.Just load and then can be ↵
↳       retrained on additional data."""
        self._initiated=True
        # This extremely hardcoded. But should be done later on to search for alternatives.
        self.models_built=True
        p=Path("./model_saves/")
        direct=[x for x in p.iterdir() if x.is_dir()]
        if self._model:
```

```
            # Think of a method to load appropriate model.
            self._model.load_weights("./model_saves/" + self._model.name + ".h5")
        else:
            for each in direct:
                model=load_model(str(each))
                model_name=each.name
                self._models_list.update({model_name:model})
                #self._models_list[model_name].load_weights("./model_saves/" + each)
        #self.compile_model()
        self._fit_done=True
        try:
            file=open("./model_config/NLP_data.json","r")
            j=json.load(file)
            file.close()
            self.model_best=j["best_model"]
        except:
            raise FileNotFoundError("Supplementary data file for NLP Core is missing. Plese check it.")

        #self.model_best="MultiAtt_cLSTM" # Hardcoded here but should be deciphered from outside file that stores the config.
        #self.model_best,_=self.choose_best_model()
        #print("NN models loaded successfully")

    #

    ########################################################
    #  Function that conditions the target for training.
    ########################################################
    # Signed Debugging -- Working Correctly.
    def prepare_target(self):
        """This is created when things are first trained.Saved and then loaded appropriately"""
        df=pd.get_dummies(self.target,dtype="int")
        return df
    #
    ########################################################
    #  EOF
    ########################################################

    ########################################################
    # Function that initiates different models depending upon the choice and
    # can be dynamically initiated
    ########################################################
    # Signed Debugging -- working correctly.
    def initiate(self,model_name=None):
        """This will actually initiatize the chosen model and make it ready for training, fitting and all kinds of functions. At the same time once the model is chosen, it is
```

```
        finalized. If we want to change our model, we have to reinitialize our entire class ↵
        and call appropriate methods.\n This method actually compiles the chosen arch and ↵
        then """
        model=self.Model_1()
        self._models_list.update({model.name : model})
        model=self.Model_2()
        self._models_list.update({model.name : model})
        model=self.Model_3()
        self._models_list.update({model.name : model})
        model=self.Model_4()
        self._models_list.update({model.name : model})
        model=self.Model_5()
        self._models_list.update({model.name : model})
        if model_name:
            self._model=self._models_list[model_name]
        self._initiated=True
        #else:
        #    self._models_list.update({"Model_1" : self.Model_1()})
        #    self._models_list.update({"Model_2" : self.Model_2()})
        #    self._models_list.update({"Model_3" : self.Model_3()})
        #    self._models_list.update({"Model_4" : self.Model_4()})
        #    self._models_list.update({"Model_5" : self.Model_5()})
        #self._classifier=self.Model_country() # Modeling other columns isn't undertaken yet.

    #
    ########################################################
    # EOF
    ########################################################

    ########################################################
    # Classifier Model for guessing the Industry - Currently on Hold
    # Theoretically, we can use the same method as the other models for target
    # column to make the model learn automatically.
    ########################################################
    #def Model_country(self):
    #    inp=k.layers.Input(shape=(12,),dtype="float") # You have to change this
    #    layer1=k.layers.Dense(12,activation=tf.nn.relu)(inp)
    #    layer2=k.layers.Dense(24,activation=tf.nn.relu)(layer1)
    #    dropout=k.layers.Dropout(0.2)(layer2)
    #    layer3=k.layers.Dense(3,activation=tf.nn.softmax)(dropout)
    #    l=k.Model(inputs=inp,outputs=layer3,name="classifier")
    #    return l
    ########################################################
    # End of Model
    ########################################################

    ########################################################
```

```
    # Function that pads sequences as per the requirement given in the module
    ########################################################
    # Signed Debugging -- Working Correctly.
    def pad_sequences(self,seq):
        """Squences are padded according to the limits as specified by the tunable parameter here."""
        return pad_sequences(seq,maxlen=self.max_len)
    #
    ########################################################
    # EOF
    ########################################################

    ########################################################
    # Model_1 Initialization  ## Tune the Model here
    ########################################################
    # Signed Debugging -- Working correctly.
    def Model_1 (self): # Model name to change according to Architecture
        """Initiating the Model here itself, this is going to help set and tune the parameters
        and models perfectly according to the situation. The Most import point this is going to do
        is set the parameters for the model."""
        embedding_dim=self.Embedding_dimensions # Default parameter for the entire engine but is tuanble according to preference.
        length_of_sequence=self.max_len
        #logic to decide the no of units based on the chosen length of sequence. But the default parameter is chosen here.
        units=[self.units,int(self.units/2),len(self.target.columns)]
        model= NN.Simple_LSTM(vocab_size=self.vocab_size,
                              embedding_dim=embedding_dim,
                              units_list=units,
                              length_of_sequence=length_of_sequence)
        # Decide this later based on
        # Setting the parameters for the model here itself
        #model.act="relu"
        #model.drop_out=0.2
        #model.lr=0.01
        #model.loss="categorical_crossentropy"
        #model.optimizer="adam"
        #model.metrics=["categorical_accuracy"]
        return model
    #
    ########################################################
    # EOF
    ########################################################
    #
    ########################################################
    # Model_2 Initialization ## Tune the model here
```

```
    ########################################################
    # Signed Debugging -- Working Correclty.
    def Model_2 (self):
        """Initiating the Model here itself, this is going to help set and tune the parameters
        and models perfectly according to the situation. The Most import point this is going ↵
↳       to do
        is set the parameters for the model."""
        embedding_dim=self.Embedding_dimensions # Default parameter for the entire engine ↵
↳       but is tuanble according to preference.
        length_of_sequence=self.max_len
        #logic to decide the no of units based on the chosen length of sequence. But the ↵
↳       default parameter is chosen here.
        units=[self.units,self.units,int(self.units/4),len(self.target.columns)]
        model= NN.Simple_BiLSTM(vocab_size=self.vocab_size,
                                embedding_dim=embedding_dim,
                                units_list=units,
                                length_of_sequence=length_of_sequence)
        # Decide this later based on
        # Setting the parameters for the model here itself
        #model.act="relu"
        #model.drop_out=0.2
        #model.lr=0.01
        #model.loss="categorical_crossentropy"
        #model.optimizer="adam"
        #model.metrics=["categorical_accuracy"]
        return model
    ########################################################
    # EOF
    ########################################################
    #
    ########################################################
    # Model_3 Initialization ## Change the model here
    ########################################################
    # Signed -- Debugging Working Correclty.
    def Model_3 (self): # Model name to change according to Architecture
        """Initiating the Model here itself, this is going to help set and tune the parameters
        and models perfectly according to the situation. The Most import point this is going ↵
↳       to do
        is set the parameters for the model."""
        embedding_dim=self.Embedding_dimensions # Default parameter for the entire engine ↵
↳       but is tuanble according to preference.
        length_of_sequence=self.max_len
        #logic to decide the no of units based on the chosen length of sequence. But the ↵
↳       default parameter is chosen here.
        units=[self.units,self.units,int(self.units/4),len(self.target.columns)]
        model= NN.SelfAtt_LSTM(vocab_size=self.vocab_size,
                                embedding_dim=embedding_dim,
```

```
                            units_list=units,
                            length_of_sequence=length_of_sequence)
        # Decide this later based on
        # Setting the parameters for the model here itself
        #model.act="relu"
        #model.drop_out=0.2
        #model.lr=0.01
        #model.loss="categorical_crossentropy"
        #model.optimizer="adam"
        #model.metrics=["categorical_accuracy"]
        #model.attention_width_given=15
        #model.attention_activation_given="sigmoid"
        return model
    #
    ######################################################
    # EOF
    ######################################################
    #
    ######################################################
    # Model_4 Initialization ## Tune the model here
    ######################################################
    # Signed -- Debugging Correctly.
    def Model_4 (self): # Model name to change according to Architecture
        """Initiating the Model here itself, this is going to help set and tune the parameters
        and models perfectly according to the situation. The Most import point this is going ↵
↳       to do
        is set the parameters for the model."""
        embedding_dim=self.Embedding_dimensions # Default parameter for the entire engine ↵
↳       but is tuanble according to preference.
        length_of_sequence=self.max_len
        #logic to decide the no of units based on the chosen length of sequence. But the ↵
↳       default parameter is chosen here.
        units=[self.units,self.units,int(self.units/4),len(self.target.columns)]
        model= NN.SelfAtt_cBiLSTM(vocab_size=self.vocab_size,
                            embedding_dim=embedding_dim,
                            units_list=units,
                            length_of_sequence=length_of_sequence)
        # Decide this later based on
        # Setting the parameters for the model here itself
        #model.act="relu"
        #model.drop_out=0.2
        #model.lr=0.01
        #model.loss="categorical_crossentropy"
        #model.optimizer="adam"
        #model.metrics=["categorical_accuracy"]
        #model.attention_width_given=15
        #model.attention_activation_given="sigmoid"
```

```
        #model.cKernel_size=4 # The Kernel size for the convolution
        #model.cFilters=200 # No of convolution filters that are placed.
        #model.cPadding="same" # padding that is requied for Convolutional layer
        return model
    #
    ######################################################
    # EOF
    ######################################################
    #
    ######################################################
    # Model_5 Initialization ## Tune the model here
    ######################################################
    # Signed -- Debugged working correctly.
    def Model_5 (self): # Model name to change according to Architecture
        """Initiating the Model here itself, this is going to help set and tune the parameters
        and models perfectly according to the situation. The Most import point this is going ↵
↳       to do
        is set the parameters for the model."""
        embedding_dim=self.Embedding_dimensions # Default parameter for the entire engine ↵
↳       but is tuanble according to preference.
        length_of_sequence=self.max_len
        #logic to decide the no of units based on the chosen length of sequence. But the ↵
↳       default parameter is chosen here.
        units=[self.units,int(self.units/4),len(self.target.columns)]
        model= NN.MultiAtt_cLSTM(vocab_size=self.vocab_size,
                                 embedding_dim=embedding_dim,
                                 units_list=units,
                                 length_of_sequence=length_of_sequence)
        # Decide this later based on
        # Setting the parameters for the model here itself
        #model.act="relu"
        #model.drop_out=0.2
        #model.lr=0.01
        #model.loss="categorical_crossentropy"
        #model.optimizer="adam"
        #model.metrics=["categorical_accuracy"]
        #model.attention_heads=3
        #model.dimension_keys=3
        #model.cKernel_size=4 # The Kernel size for the convolution
        #model.cFilters=200 # No of convolution filters that are placed.
        #model.cPadding="same" # padding that is requied for Convolutional layer
        return model
    #
    ######################################################
    # EOF
    ######################################################
    #
```

```
    ########################################################
    # Function Initiating and building the models and their names
    ########################################################
    # Signed  Debugged -- Working Correctly.
    def build_models(self):
        """This function will build models and keep the models along with their
        architectures aside for selection and
        training later on. Models are compiled only once the selection of the architecture
        is done. Building of models
        happen automatically if its a train=true is chose.
        Actual input shapes are left to the fit time, based on user inputs, or depending
        upon the dataset provided."""
        if self._model:
            self._model.build(input_shape=(None,None,)) #
        else:
            for each in self._models_list:
                model=self._models_list[each]
                model.build(input_shape=(None,None,)) # Can be instantiaed later on when
                things get going, to actually fix the size later on.
        self.models_built=True
    #
    ########################################################
    # EOF
    ########################################################

    ########################################################
    # Function showing the list of available Models
    # If initiali
    ########################################################
    # Signed Debugged -- Working Correctly.
    def show_models_names(self):
        """Shows the list of available Models"""
        if self.models_built:
            print("Please select from among below models with appropriate call function\n")
            x=["{}".format(each) for each in self._models_list]
            for each in x:
                print(each)
            print("If you want to see Network architecture, use the method
            \"show_model_arch()\" for showing the architecture summary")
            return
        else :
            raise NotImplementedError("The models are not yet built")
    #
    ########################################################
    # EOF
    ########################################################
```

```
    ######################################################
    # Function showing the different Model Architectures
    ######################################################
    # Signed Debugged -- Working Correctly.
    def show_model_arch(self,mod):
        """ Different model architectures as chosen are represented in this function.
        If model name is not in the list, it asks the user to check the spelling of the chosen model.
        If the model is in the list, the architecture summary of the model is returned. It can be fed to print function"""
        if self.models_built:
            try:
                assert mod in self._models_list.keys()
                return self._models_list[mod].summary()
            except:
                print(" Sorry ! Please check the spelling of your input or the name. If its right, please check if input is the key, not the model name")
                raise ValueError("Model you choose is not in the Architectures available")
        else:
            raise ValueError("The Model you choose haven't been built yet. Please build them first or follow the guidelines")
    #
    ######################################################
    # EOF
    ######################################################
    #
    ######################################################
    # List of functions for classifier model, handled separately, but not implemented here.
    ######################################################
    #
    #
    #def classifier_model_summary(self):
    #    return self._classifier.summary()
    #
    #def classifier_model_history(self):
    #    return self._classifer.history()
    #
    #def classifier_model_fit(self,*args,**kwargs):
    #    if self._initiated:
    #        self._classifier.fit(*args,**kwargs) # This needs modification
    #
    #def classifier_model_compile(self,*args,**kwargs):
    #    if self._initiated:
    #        self._classifier.compile(*args,**kwargs) # This needs modification
    #
    ######################################################
```

```
    # EOF
    ######################################################
    #
    # There should be a separate train function that will automatically
    # call all these funcitons with appropriate parameters.
    ######################################################
    # Important Training related functions.
    ######################################################
    #
    # Should write the script to fit the model with the data already here.
    # Signed Debugging -- Working Done perfectly.
    def fit(self,*args,**kwargs):
        """This is where the actual fitting is done. Batch size dimensions and """
        X_train=self.data2
        y_train=self.target
        callback=self.call_back
        if self._initiated:
            if self._model:
                self._model.fit(x=X_train,y=y_train,callbacks=callback,*args,**kwargs)
            else:
                for mod in self._models_list:
                    #print("New Model Started :",mod)
                    #x=copy.copy(X_train)
                    #y=copy.copy(y_train)
                    #print("X Shape : ",x.shape)
                    #print("y Shape : ",y.shape)
                    model=self._models_list[mod]
                    opt=copy.copy(self.optimizer_chosen)
                    los=copy.copy(self.loss_chosen)
                    met=copy.copy(self.metric_chosen)
                    if not self.model_compiled:
                        try:
                            model.compile(optimizer=opt,loss=los,metrics=[met])
                        except:
                            pass
                    try:
                        #model.fit(x=x,y=y,epochs=self.epoch,*args,**kwargs)
                        model.fit(x=X_train,y=y_train,*args,**kwargs)
                    except:
                        print(mod)
                    else:
                        print("Finished Building Model :", mod)
                self._fit_done=True
        else:
            print("Please select an appropriate model architecture")
            raise ValueError("Model Not Selected")
        return
```

```
    #
    #def build_sup(self,*args,**kwargs):
    #    if self._initiated:
    #        self._model_chosen.build(*args,**kwargs)
    #    else:
    #        print("Please select an appropriate model architecture")
    #        raise ValueError("Model Not Selected")
    #
    # Signed -- Debugging done - working perfectly.
    def predict(self,in_array,*args,**kwargs):
        """Parameters specifically chosen for predict paramters."""
        # Note very important point to reshape your model appropriately.
        #print("The arrived shape is :",in_array.shape)
        #x_in=np.expand_dims(in_array,axis=0)
        x_in=in_array
        if self._fit_done:
            try:
                if self._model:
                    l=self._model.predict(*args,**kwargs)
                else:
                    l=[]
                    for each in self._models_list:
                        model=self._models_list[each]
                        #print(model.summary())
                        try:
                            l.append(model.predict(x=x_in,*args,**kwargs))
                        except:
                            pass
                return l
            except:
                raise ValueError("The proper method is not given")
        else:
            raise NotImplementedError("The fitting hasn't been done. So, you can't predict.")
    # Signed -- Debugging done - working Perfectly. But Don't use it before. Use it in ↵
↳   connection with fit function.
    def compile_model(self,*args,**kwargs):
        """This will compile the NN Model. It will over ride certain setting like optimizer ↵
↳       and all, but usually shouldn't be a
        problem while training."""
        opt=self.optimizer_chosen
        los=self.loss_chosen
        met=self.metric_chosen
        #met="categorical_crossentropy"
        try:
            assert self.models_built
        except:
            print("Please build the model properly before compiling it")
```

```
            raise ValueError("Model Not Selected")
        else:
            if self._model:
                self._model.compile(optimizer=opt,loss=los,metrics=[met],*args,**kwargs)

            else:
                for each in self._models_list:
                    opt=copy.copy(self.optimizer_chosen)
                    los=copy.copy(self.loss_chosen)
                    met=copy.copy(self.metric_chosen)
                    model=self._models_list[each]
                    model.compile(optimizer=opt,loss=los,metrics=[met],*args,**kwargs)
                    print(model.summary())
        self.model_compiled=True
        return
    #
    def optimize(self):
        """This function will optimize the model in case a given model is selected. If not
        it will simply raise an error"""
        try:
            assert self._model
            print("Model is being optimized here")
        except:
            raise NotImplementedError("It is not possible to optimize all the given models
            here. Please select one to proceed with optimization.")
        else:
            pass
            # Implement the code to optimize the model here.

    #def get_config(self):
    #    if not self._model_chosen == None:
    #        try:
    #            #j=super(NN_NLP,self).get_config()
    #            return self._model_chosen.get_config()
    #        except:
    #            raise ValueError("This is presently not supported")
    #    else:
    #        raise ValueError("Please Choose a Model First")
    #

    ######################################################
    # End of Overload Methods
    ######################################################

#%%
```

d:\Work_folders\code\python-Spyder\Shravani-Chatbot\NN_models.py

```
# -*- coding: utf-8 -*-
"""
Created on Fri Sep 24 00:28:17 2021
This is the core file where all the models are stored as they are.

@author: Sheshank_Joshi
"""
#%%
import tensorflow as tf
#from tensorflow.keras.models import Model,Sequential
from tensorflow.keras.layers import Dense, Embedding, LSTM, SpatialDropout1D, Bidirectional, MultiHeadAttention, Conv1D, Dropout
from tensorflow.keras import regularizers as reg
from keras_self_attention import SeqSelfAttention
#from tensorflow.keras.optimizers import Adam
#from tensorflow.keras.losses import CategoricalCrossentropy
#from tensorflow.keras.metrics import CategoricalAccuracy
#from tensorflow.keras import models
#%%
class Simple_LSTM(tf.keras.Model):
    """Default values are set here based on testing stage. But, they can always be set different values by the
    management engine."""
    act="relu" # Activation layer that can be used for various layers
    _sm="softmax" # Softmax input for last layer. This can't be changed.
    drop_out=0.2 # Resettable dropout for the entire class
    lr=0.01 # Predefined learning rate can be tuned
    los="categorical_crossentropy" # Loss function can be returend later on
    opt="adam" # Optimizer function that can be retuned later on
    met=["categorical_accuracy"] # List of metrics that can be monitored from manager side.
    def __init__(self, vocab_size, embedding_dim, units_list, length_of_sequence):
        super(Simple_LSTM, self).__init__(name="Simple_LSTM")
        self.units = units_list
        #self.callback=callback
        self.embedding = Embedding(vocab_size, embedding_dim,trainable=True,input_length=length_of_sequence)

        self.layer1=LSTM(self.units[0],activation=self.act,recurrent_dropout=0.2,dropout=self.drop_out)

        self.layer2=Dense(self.units[1],activation=self.act,kernel_regularizer=reg.L1(l1=self.lr))

        self.layer3=Dense(self.units[2],activation=self._sm,kernel_regularizer=reg.L2(l2=self.lr))

    def call(self, inputs):
```

```
        x = self.embedding(inputs)
        x = self.layer1(x)
        x=self.layer2(x)
        outputs=self.layer3(x)
        return outputs


#%%
#k=Simple_BiLSTM(25,25,[12,15,10,5],4)
#%%
class Simple_BiLSTM(tf.keras.Model):
    """This model has total 3 layers. Total Number of units of each layer is specified in units_list, in appropriate order. If not enough parameters are not provided,
    it will throuigh out of index error. So, be careful."""
    act="relu" # Activation layer that can be used for various layers
    _sm="softmax" # Softmax input for last layer. This can't be changed.
    drop_out=0.2 # Resettable dropout for the entire class
    lr=0.01 # Predefined learning rate can be tuned
    loss="categorical_crossentropy" # Loss function can be returend later on
    optimizer="adam" # Optimizer function that can be retuned later on
    metrics=["categorical_accuracy"] # List of metrics that can be monitored from manager side.
    def __init__(self, vocab_size, embedding_dim, units_list, length_of_sequence):
        super(Simple_BiLSTM, self).__init__(name="Simple_BiLSTM")
        self.units = units_list
        #self.callback=callback
        self.embedding = Embedding(vocab_size, embedding_dim,trainable=True,input_length=length_of_sequence)
        self.drop1=SpatialDropout1D(self.drop_out)

        self.layer1=Bidirectional(LSTM(self.units[0],dropout=self.drop_out,return_sequences=True,activation=self.act,recurrent_dropout=0.2))

        self.layer2=LSTM(self.units[1],activation=self.act,dropout=self.drop_out,recurrent_dropout=0.2)

        self.layer3=Dense(self.units[2],activation=self.act,kernel_regularizer=reg.L1(l1=self.lr))

        self.layer4=Dense(self.units[3],activation=self._sm,kernel_regularizer=reg.L1L2(l1=self.lr,l2=self.lr))

    def call(self, inputs,training=False):
        x = self.embedding(inputs)
        #if training:
        x=self.drop1(x)
        x = self.layer1(x)
        x=self.layer2(x)
```

```
        x=self.layer3(x)
        outputs=self.layer4(x)
        return outputs
#%%
#k=Simple_BiLSTM(25,25,[12,15,10,5],4)
#%%
class SelfAtt_LSTM(tf.keras.Model):
    act="relu" # Activation layer that can be used for various layers
    _sm="softmax" # Softmax input for last layer. This can't be changed.
    drop_out=0.2 # Resettable dropout for the entire class
    lr=0.01 # Predefined learning rate can be tuned
    los="categorical_crossentropy" # Loss function can be returend later on
    opt="adam" # Optimizer function that can be retuned later on
    met=["categorical_accuracy"] # List of metrics that can be monitored from manager side.
    attention_width_given=15
    attention_activation_given="sigmoid"
    def __init__(self, vocab_size, embedding_dim, units_list, length_of_sequence):
        super(SelfAtt_LSTM, self).__init__(name="SelfAtt_LSTM")
        self.units = units_list
        #self.callback=callback
        self.embedding = Embedding(vocab_size, ↵
↳       embedding_dim,trainable=True,input_length=length_of_sequence)
        self.drop1=SpatialDropout1D(self.drop_out)
        ↵
↳       self.layer1=Bidirectional(LSTM(self.units[0],activation=self.act,return_sequences=True↵
↳       ,recurrent_dropout=0.2))
        ↵
↳       self.att=SeqSelfAttention(attention_width=self.attention_width_given,attention_activat↵
↳       ion=self.attention_activation_given)
        ↵
↳       self.layer2=LSTM(self.units[1],activation=self.act,dropout=self.drop_out,recurrent_dro↵
↳       pout=0.2)
        ↵
↳       self.layer3=Dense(self.units[2],activation=self.act,kernel_regularizer=reg.L1(l1=self.↵
↳       lr))
        ↵
↳       self.layer4=Dense(self.units[3],activation=self._sm,kernel_regularizer=reg.L1L2(l1=sel↵
↳       f.lr,l2=self.lr))

    def call(self, inputs,training=False):
        x = self.embedding(inputs)
        if training:
            x=self.drop1(x,training=training)
        x=self.att(x)
        x = self.layer1(x)
        x=self.layer2(x)
        x=self.layer3(x)
```

```
        outputs=self.layer4(x)
        return outputs
#%%
#k=SelfAtt_LSTM(25,25,[12,15,10,5],4)
#%%
class SelfAtt_cBiLSTM(tf.keras.Model):
    act="relu" # Activation layer that can be used for various layers
    _sm="softmax" # Softmax input for last layer. This can't be changed.
    drop_out=0.2 # Resettable dropout for the entire class
    lr=0.01 # Predefined learning rate can be tuned
    los="categorical_crossentropy" # Loss function can be returend later on
    opt="adam" # Optimizer function that can be retuned later on
    met=["categorical_accuracy"] # List of metrics that can be monitored from manager side.
    attention_width_given=15 # The sequence length to which attention is applied.
    attention_activation_given="sigmoid" #
    cKernel_size=4 # The Kernel size for the convolution
    cFilters=200 # No of convolution filters that are placed.
    cPadding="same" # padding that is requied for Convolutional layer
    #
    def __init__(self, vocab_size, embedding_dim, units_list, length_of_sequence):
        super(SelfAtt_cBiLSTM, self).__init__(name="SelfAtt_cBiLSTM")
        self.units = units_list
        #self.callback=callback
        self.embedding = Embedding(vocab_size, ↵
↳       embedding_dim,trainable=True,input_length=length_of_sequence)
        self.drop1=SpatialDropout1D(self.drop_out)
        self.conv=Conv1D(filters=self.cFilters,kernel_size=self.cKernel_size,padding='same')
        self.drop2=Dropout(self.drop_out)
        ↵
↳       self.layer1=Bidirectional(LSTM(self.units[0],dropout=self.drop_out,activation=self.act↵
↳       ,return_sequences=True,recurrent_dropout=0.2))
        ↵
↳       self.att=SeqSelfAttention(attention_width=self.attention_width_given,attention_activat↵
↳       ion=self.attention_activation_given)
        ↵
↳       self.layer2=LSTM(self.units[1],activation=self.act,dropout=self.drop_out,recurrent_dro↵
↳       pout=0.4)
        ↵
↳       self.layer3=Dense(self.units[2],activation=self.act,kernel_regularizer=reg.L1(l1=self.↵
↳       lr))
        ↵
↳       self.layer4=Dense(self.units[3],activation=self._sm,kernel_regularizer=reg.L2(l2=self.↵
↳       lr))

    def call(self, inputs,training=False):
        x = self.embedding(inputs)
        if training:
```

```
            x=self.drop1(x,training=training)
        x=self.conv(x)
        if training:
            x=self.drop2(x,training=training)
        x=self.layer1(x)
        x=self.att(x)
        x=self.layer2(x)
        x=self.layer3(x)
        outputs=self.layer4(x)
        return outputs
#%%

#%%
class MultiAtt_cLSTM(tf.keras.Model):
    act="relu" # Activation layer that can be used for various layers
    _sm="softmax" # Softmax input for last layer. This can't be changed.
    drop_out=0.2 # Resettable dropout for the entire class
    lr=0.01 # Predefined learning rate can be tuned
    los="categorical_crossentropy" # Loss function can be returend later on
    opt="adam" # Optimizer function that can be retuned later on
    met=["categorical_accuracy"] # List of metrics that can be monitored from manager side.
    attention_heads=3 # The number of heads for Multihead-Attention
    dimension_keys=3 # The number of key dimensions for the Multiple heads.
    cKernel_size=4 # The Kernel size for the convolution
    cFilters=100 # No of convolution filters that are placed.
    cPadding="same" # padding that is requied for Convolutional layer
    #
    def __init__(self, vocab_size, embedding_dim, units_list, length_of_sequence):
        super(MultiAtt_cLSTM, self).__init__(name="MultiAtt_cLSTM")
        self.units = units_list
        self.embedding = Embedding(vocab_size, ↵
↳       embedding_dim,trainable=True,input_length=length_of_sequence)
        #self.drop1 = SpatialDropout1D(self.drop_out)
        self.conv = Conv1D(filters=self.cFilters,kernel_size=self.cKernel_size,padding='same')
        ↵
↳       self.mAtt=MultiHeadAttention(num_heads=self.attention_heads,key_dim=self.dimension_key↵
↳       s)
        #self.drop2 = Dropout(self.drop_out)
        ↵
↳       #self.layer1=Bidirectional(LSTM(self.units[0],return_sequences=True,activation=self.re↵
↳       lu,recurrent_dropout=0.2))
        ↵
↳       #self.att=SeqSelfAttention(attention_width=self.attention_width_given,attention_activa↵
↳       tion=self.attention_activation_given)
        ↵
↳       self.layer1=LSTM(self.units[0],activation=self.act,dropout=self.drop_out,recurrent_dro↵
↳       pout=0.2)
```

```
        ↵
↳       self.layer2=Dense(self.units[1],activation=self.act,kernel_regularizer=reg.L1(l1=self.↵
↳       lr))
        ↵
↳       self.layer3=Dense(self.units[2],activation=self._sm,kernel_regularizer=reg.L2(l2=self.↵
↳       lr))
    #
    def call(self, inputs,training=False):
        x=self.embedding(inputs)
        x=self.conv(x)
        x=self.mAtt(x,x)
        x=self.layer1(x)
        x=self.layer2(x)
        outputs=self.layer3(x)
        return outputs
#%%
#k=MultiAtt_cLSTM(25,25,[12,15,10,5],4)
#%%
```

d:\Work_folders\code\python-Spyder\Shravani-Chatbot\supervised_core_manager.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Oct  3 15:20:46 2021

@author: Sheshank_Joshi
"""
#%%
#import imblearn

from sklearn.decomposition import PCA
import pandas as pd
import supervised_models as sm
from sklearn.tree import DecisionTreeRegressor
import numpy as np
import pickle as pkl
from pathlib import Path
#%%
class sup_manager(sm.sup_models):
    """This manages all the supervised learning models placed in a separate file of supervised learning
    models."""
    saved_file_name="sup_learn_models.pkl"
    def __init__(self,X=None,y=None,train=False,auto=False):
        super(sup_manager,self).__init__()
        self.no_of_components=None
        self.minimum_threshold=0.005 # This is the variance ratio to be used while doing feature engineering with PCA
        self.percentage=None
        self.scores={}
        self.X_train=None
        self.y_train=None
        self.pca=None
        self.models=None

        if train:
            if type(X)==type(pd.DataFrame()):
                self.X_train=X
                self.y_train=y
            else:
                raise ValueError("The Data type provided is not the data type that is required.")
            if auto:
                self.PCA_transform_features()
                self.fit()
                self.save()
        else:
            self.Models_list={}
```

```
            sup_manager.load(self)
            print("Supervised Models are Loaded Successfully")
        #self.X_train,self.y_train=self.shape_resampling(self.X_train,self.y_train)
    #
    # Debugging done -- Working Perfectly
    def save(self):
        """This can be used to save the models. But, it will not save the data, it will only save the models."""
        #self.X_train=None
        #self.y_train=None
        self.models=list(self.Models_list.keys())
        try:
            f=open("./model_saves/"+self.saved_file_name,"wb")
            pkl.dump(self,f)
            f.close()
            for each in self.Models_list:
                model=self.Models_list[each]
                file=open("./model_saves/"+each+".sav","wb")
                pkl.dump(model,file)
                file.close()
            #print("Supervised Models are saved")
        except:
            raise FileNotFoundError("The Given file can't be saved because the structure doesn't exist.")
    #
    #Signed -- Debugging done -- working perfectly.
    @classmethod
    def load(cls,self):
        """This will load the model from the saved files and makes it ready for prediciton, though the original data will be gone."""
         #print("--------------Before loading----------------")
        f=open("./model_saves/"+cls.saved_file_name,"rb")
        obj=pkl.load(f)
        f.close()
        #models=
        #print(dir(obj))
        #print(obj._fit_done)
        #print("Total Vocabulary Size loaded :",obj.model["l_1"].vocab._len)
        #print("The length of the Vocabulary is :",len(obj.vocab))
        #print(obj.avg_scores)
        self.__dict__=obj.__dict__.copy()
        self.Models_list={}
        p=Path("./model_saves/")
        direct=[x.name for x in p.iterdir() if not x.is_dir()]
        #print(direct)
        for each in self.models:
            file_name=each+".sav"
```

```
            if file_name in direct:
                file=open("./model_saves/"+file_name,"rb")
                mod=pkl.load(file)
                self.Models_list.update({each:mod})
                file.close()
            else:
                print("The model missing for loading is :",each)
        # Write script to load each of the fitted model into the dictionary object.
        # copying the original data into memory
    #
    # Signed -- Debugging done -- Working perfectly
    def pca_tuning(self,X_train):
        parameters_to_choose=range(len(X_train.columns))
        decision_maker={}
        for i in parameters_to_choose:
            pca=PCA(n_components=i)
            pca.fit(X_train)
            #print("Checkpoint")
            pca_variance=pca.explained_variance_ratio_.sum()
            decision_maker.update({i:pca_variance})
            #print(decision_maker)
        decision_maker=pd.Series(decision_maker).diff()
        out=decision_maker[decision_maker>self.minimum_threshold].index.max()
        return out
    #
    # Signed -- Debugging Done -- working Perfectly.
    def PCA_transform_features(self):
        # We can tune this later for whatever value we want for components that can be ↵
↳       neglected for their influence
        n=0 #Initialize the number of features to be 0.
        #print(no_of_components)
        if self.no_of_components==None:
            n=self.pca_tuning(self.X_train)
            #print("Here in no components")
            #print(n)
        else:
            n=self.no_of_components
        #print("No of components selected is :",n)
        pca=PCA(n_components=n,random_state=self.random_state)
        pca.fit(self.X_train)
        self.pca=pca
        #print(pca.explained_variance_ratio_)
        self.X_train=pd.DataFrame(pca.transform(self.X_train))
    #
    # Debugging done -- working Perfectly.
    def fit(self):
        """This fits into all the models the given training and testing data."""
```

```
        for each in self.Models_list:
            try:
                model=self.Models_list[each]
                #print(self.X_train,self.y_train)
                model.fit(self.X_train,self.y_train)
                self.scores.update({each:model.score(self.X_train,self.y_train)})
            except:
                print("Problem with fitting the model :",each)
    #
    # Debugging done -- working perfectly.
    def predict(self,outside_data):
        """All the models' predictions are returned. These can be used further for analysis."""
        predictions=[]
        data=self.pca.transform(outside_data)
        #data=np.expand_dims(outside_data,axis=0)
        #print(data)
        #print(data.shape)
        for each_model in self.Models_list:
            model=self.Models_list[each_model]
            try:
                pred=model.predict(data)[0]
                #print(pred)
                predictions.append(pred)
            except:
                print("Something wrong with prediction for the model :",each_model)
        return predictions
    #
    #
    def Decisiontree_features(self,percentage=None):
        if percentage==None:
            n_features_chosen=self.X_train.shape[1]*10/100
        else:
            n_features_chosen=self.X_train.shape[1]*percentage/100
        k=n_features_chosen
        regressor = DecisionTreeRegressor(random_state=self.random_state, max_depth=5)
        regressor.fit(self.X_train,self.y_train)
        feature_importances = regressor.feature_importances_
        feature_names = self.X_train.columns
        top_k_idx = (feature_importances.argsort()[-k:][::-1])
        #print(feature_names[top_k_idx], feature_importances)
        return self.X_train[feature_names[top_k_idx]]
    #
#%%
```

d:\Work_folders\code\python-Spyder\Shravani-Chatbot\supervised_models.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Oct  3 15:21:10 2021

@author: Sheshank_Joshi
"""
#%%
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import NuSVC
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
import json
#%%
class sup_models():
    random_state=4
    Models_list={}
    #
    LR_model = LogisticRegression()
    Models_list.update({"LogisticRegression":LR_model})
    #
    NB_model = GaussianNB()
    Models_list.update({"NaiveBayes":NB_model})
    KNN = KNeighborsClassifier()
    Models_list.update({"KNN":KNN})
    #
    support_vector = SVC()
    Models_list.update({"support_vector":support_vector})
    #
    decision_tree=DecisionTreeClassifier()
    Models_list.update({"decision_tree":decision_tree})
    #
    bagging=BaggingClassifier(random_state=random_state)
    Models_list.update({"bagging":bagging})
    #
    adaboost=AdaBoostClassifier(n_estimators=100,random_state=random_state)
    Models_list.update({"adaboost":adaboost})
    #
    gradientboost = GradientBoostingClassifier(random_state=random_state)
    Models_list.update({"gradientboost":gradientboost})
```

```
    #
    random_forest = RandomForestClassifier(random_state=random_state)
    Models_list.update({"random_forest":random_forest})
    #
    Mlpc = MLPClassifier()
    Models_list.update({"MLPC":Mlpc})
    #
    Nusvc=NuSVC()
    Models_list.update({"NuSVC":Nusvc})
    #
    parameters=None
    def __init_(self):
        try:
            file=open("./param_saves/sup_model_refer_params.json","r")
            self.parameters=json.load(file)
            file.close()
        except:
            file.close()
            raise Warning("The parameters file is missing, so going with unkonwn default ↵
↳           parameters. You don't be able to tune.")
    #
    def parameter_generator(self,model_name):
        """Will prepare the parameter search engine for the model and the best parameters."""
        gs = GridSearchCV(model_name,param_grid=self.parameters[model_name],cv=10)
        return gs
    #
    def tune(self,X_train,y_train):
        """This will tune the models appropriately for the object"""
        best_param_gridsearch={}
        for each in self.Models_list:
            gs=self.parameter_generator(each)
            gs.fit(X_train,y_train)
            best_param_gridsearch.update({each:gs.best_params_})
            model=self.Models_list[each]
            model.set_params(best_param_gridsearch)
        try:
            file=open("./param_saves/sup_model_best_params.json","w")
            json.dump(best_param_gridsearch,file)
            file.close()
        except:
            print("The appropriate folder hasn't been found. Hence saving the parameters is ↵
↳           not implemented.")
    #
```

d:\Work_folders\code\python-Spyder\Shravani-Chatbot\supervised_models.py

```
#%%

#%%

#%%
```