Chatbot:

import io

import random

import string # to process standard python strings

import warnings

import numpy as np

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

import warnings

warnings.filterwarnings('ignore')

pip install nltk

import nltk

from nltk.stem import WordNetLemmatizer

nltk.download('popular', quiet=True) # for downloading packages

nltk.download('punkt') # first-time use only

nltk.download('wordnet') # first-time use only

f=open('chatbot.txt','r',errors = 'ignore')

raw=f.read()

raw = raw.lower()# converts to lowercase

sent\_tokens = nltk.sent\_tokenize(raw)# converts to list of sentences

word\_tokens = nltk.word\_tokenize(raw)# converts to list of words

lemmer = nltk.stem.WordNetLemmatizer()

def LemTokens(tokens):

return [lemmer.lemmatize(token) for token in tokens]

remove\_punct\_dict = dict((ord(punct), None) for punct in string.punctuation)

def LemNormalize(text):

return LemTokens(nltk.word\_tokenize(text.lower().translate(remove\_punct\_dict)))

GREETING\_INPUTS = ("hello", "hi", "greetings", "sup", "what's up","hey",)

GREETING\_RESPONSES = ["hi", "hey", "\*nods\*", "hi there", "hello", "I am glad! You are talking to me"]

def greeting(sentence):

for word in sentence.split():

if word.lower() in GREETING\_INPUTS:

return random.choice(GREETING\_RESPONSES)

def response(user\_response):

robo\_response=''

sent\_tokens.append(user\_response)

TfidfVec = TfidfVectorizer(tokenizer=LemNormalize, stop\_words='english')

tfidf = TfidfVec.fit\_transform(sent\_tokens)

vals = cosine\_similarity(tfidf[-1], tfidf)

idx=vals.argsort()[0][-2]

flat = vals.flatten()

flat.sort()

req\_tfidf = flat[-2]

if(req\_tfidf==0):

robo\_response=robo\_response+"I am sorry! I don't understand you"

return robo\_response

else:

robo\_response = robo\_response+sent\_tokens[idx]

return robo\_response

flag=True

print("ROBO: My name is Robo. I will answer your queries about Chatbots. If you want to exit, type Bye!")

while(flag==True):

user\_response = input()

user\_response=user\_response.lower()

if(user\_response!='bye'):

if(user\_response=='thanks' or user\_response=='thank you' ):

flag=False

print("ROBO: You are welcome..")

else:

if(greeting(user\_response)!=None):

print("ROBO: "+greeting(user\_response))

else:

print("ROBO: ",end="")

print(response(user\_response))

sent\_tokens.remove(user\_response)

else:

flag=False

print("ROBO: Bye! take care..")

--------------------------------------------------------------------------------------------------------------------------------------Expert System

import streamlit as st  
import yfinance as yf  
import pandas as pd  
  
st.text('Ankur Singh Solanki')  
st.title('Stock Trading Market')  
  
tickers = ('TSLA','AAPL','MSFT','BTC-USD','ETH-USD')  
  
dropdown = st.multiselect('Pick your assets',tickers)  
  
start = st.date\_input('Start',value = pd.to\_datetime('2021-01-01'))  
end = st.date\_input('End',value = pd.to\_datetime('today'))  
  
def relativeret(df):  
 rel = df.pct\_change()  
 cumret = (1+rel).cumprod()-1  
 cumret = cumret.fillna(0)  
 return cumret  
  
if len(dropdown) > 0:  
 data = yf.download(dropdown,start,end)['Adj Close']  
 df = relativeret(data)  
 st.line\_chart(df)

N-queen bb

#include <iostream>

# include <string.h>

**using** **namespace** std;

#define N 8

**void** printSolution(**int** board[N][N])

{

**for** (**int** i = 0; i < N; i++)

    {

**for** (**int** j = 0; j < N; j++)

            cout << " "<< board[i][j];

        cout << "\n";

    }

}

**bool** isSafe(**int** row, **int** col, **int** slashCode[N][N],

**int** backslashCode[N][N], **bool** rowLookup[],

**bool** slashCodeLookup[], **bool** backslashCodeLookup[] )

{

**if** (slashCodeLookup[slashCode[row][col]] ||

        backslashCodeLookup[backslashCode[row][col]] ||

        rowLookup[row])

**return** **false**;

**return** **true**;

}

**bool** solveNQueensUtil(**int** board[N][N], **int** col,

**int** slashCode[N][N], **int** backslashCode[N][N],

**bool** rowLookup[N],

**bool** slashCodeLookup[],

**bool** backslashCodeLookup[] )

{

**if** (col >= N)

**return** **true**;

**for** (**int** i = 0; i < N; i++)

    {

**if** ( isSafe(i, col, slashCode,

                    backslashCode, rowLookup,

          slashCodeLookup, backslashCodeLookup) )

        {

            board[i][col] = 1;

            rowLookup[i] = **true**;

            slashCodeLookup[slashCode[i][col]] = **true**;

            backslashCodeLookup[backslashCode[i][col]] = **true**;

**if** ( solveNQueensUtil(board, col + 1,

                                  slashCode, backslashCode,

             rowLookup, slashCodeLookup, backslashCodeLookup) )

**return** **true**;

            board[i][col] = 0;

            rowLookup[i] = **false**;

            slashCodeLookup[slashCode[i][col]] = **false**;

            backslashCodeLookup[backslashCode[i][col]] = **false**;

        }

    }

**return** **false**;

}

**bool** solveNQueens()

{

**int** board[N][N];

**memset**(board, 0, **sizeof** board);

**int** slashCode[N][N];

**int** backslashCode[N][N];

**bool** rowLookup[N] = {**false**};

**bool** slashCodeLookup[2\*N - 1] = {**false**};

**bool** backslashCodeLookup[2\*N - 1] = {**false**};

**for** (**int** r = 0; r < N; r++)

**for** (**int** c = 0; c < N; c++) {

          slashCode[r] = r + c,

            backslashCode[r] = r - c + 7;

        }

**if** (solveNQueensUtil(board, 0,

                          slashCode, backslashCode,

      rowLookup, slashCodeLookup, backslashCodeLookup) ==

**false** )

    {

        cout << "Solution does not exist";

**return** **false**;

    }

    printSolution(board);

**return** **true**;

}

**int** main()

{

    solveNQueens();

**return** 0;

}

N queen backtracking

#include <bits/stdc++.h>

#define N 4

**using** **namespace** std;

**void** printSolution(**int** board[N][N])

{

**for** (**int** i = 0; i < N; i++) {

**for** (**int** j = 0; j < N; j++)

            cout << " " << board[i][j] << " ";

**printf**("\n");

    }

}

**bool** isSafe(**int** board[N][N], **int** row, **int** col)

{

**int** i, j;

**for** (i = 0; i < col; i++)

**if** (board[row][i])

**return** **false**;

**for** (i = row, j = col; i >= 0 && j >= 0; i--, j--)

**if** (board[i][j])

**return** **false**;

**for** (i = row, j = col; j >= 0 && i < N; i++, j--)

**if** (board[i][j])

**return** **false**;

**return** **true**;

}

**bool** solveNQUtil(**int** board[N][N], **int** col)

{

**if** (col >= N)

**return** **true**;

**for** (**int** i = 0; i < N; i++) {

**if** (isSafe(board, i, col)) {

            board[i][col] = 1;

**if** (solveNQUtil(board, col + 1))

**return** **true**;

            board[i][col] = 0;

        }

    }

**return** **false**;

}

**bool** solveNQ()

{

**int** board[N][N] = { { 0, 0, 0, 0 },

                        { 0, 0, 0, 0 },

                        { 0, 0, 0, 0 },

                        { 0, 0, 0, 0 } };

**if** (solveNQUtil(board, 0) == **false**) {

        cout << "Solution does not exist";

**return** **false**;

    }

    printSolution(board);

**return** **true**;

}

**int** main()

{

    solveNQ();

**return** 0;

}

GraphColoring

#include<bits/stdc++.h>

**using** **namespace** std;

#define V 4

**void** printSolution(**int** color[]);

**bool** isSafe(**bool** graph[V][V], **int** color[])

{

**for** (**int** i = 0; i < V; i++)

**for** (**int** j = i + 1; j < V; j++)

**if** (graph[i][j] && color[j] == color[i])

**return** **false**;

**return** **true**;

}

**bool** graphColoring(**bool** graph[V][V], **int** m, **int** i,

**int** color[V])

{

**if** (i == V) {

**if** (isSafe(graph, color)) {

            printSolution(color);

**return** **true**;

        }

**return** **false**;

    }

**for** (**int** j = 1; j <= m; j++) {

        color[i] = j;

**if** (graphColoring(graph, m, i + 1, color))

**return** **true**;

        color[i] = 0;

    }

**return** **false**;

}

**void** printSolution(**int** color[])

{

    cout << "Solution Exists:" " Following are the assigned colors \n";

**for** (**int** i = 0; i < V; i++)

        cout << "  " << color[i];

    cout << "\n";

}

**int** main()

{

**bool** graph[V][V] = {

        { 0, 1, 1, 1 },

        { 1, 0, 1, 0 },

        { 1, 1, 0, 1 },

        { 1, 0, 1, 0 },

    };

**int** m = 3;

**int** color[V];

**for** (**int** i = 0; i < V; i++)

        color[i] = 0;

**if** (!graphColoring(graph, m, 0, color))

        cout << "Solution does not exist";

**return** 0;

}

Graph Coloring BB

#include <bits/stdc++.h>

#include <iostream>

**using** **namespace** std;

**class** node

{

**public**:

**int** color = 1;

    set<**int**> edges;

};

**int** canPaint(vector<node>& nodes, **int** n, **int** m)

{

    vector<**int**> visited(n + 1, 0);

**int** maxColors = 1;

**for** (**int** sv = 1; sv <= n; sv++)

    {

**if** (visited[sv])

**continue**;

        visited[sv] = 1;

        queue<**int**> q;

        q.push(sv);

**while** (!q.empty())

        {

**int** top = q.front();

            q.pop();

**for** (**auto** it = nodes[top].edges.begin();

                 it != nodes[top].edges.end(); it++)

            {

**if** (nodes[top].color == nodes[\*it].color)

                    nodes[\*it].color += 1;

                maxColors

                    = max(maxColors, max(nodes[top].color,

                                         nodes[\*it].color));

**if** (maxColors > m)

**return** 0;

**if** (!visited[\*it]) {

                    visited[\*it] = 1;

                    q.push(\*it);

                }

            }

        }

    }

**return** 1;

}

**int** main()

{

**int** n = 4;

**bool** graph[n][n] = {

     { 0, 1, 1, 1 },

     { 1, 0, 1, 0 },

     { 1, 1, 0, 1 },

     { 1, 0, 1, 0 }};

**int** m = 3;

      vector<node> nodes(n + 1);

**for** (**int** i = 0; i < n; i++)

      {

**for**(**int** j =0;j<n;j++)

         {

**if**(graph[i][j])

             {

                  nodes[i].edges.insert(i);

                  nodes[j].edges.insert(j);

              }

         }

      }

        cout << canPaint(nodes, n, m);

        cout << "\n";

**return** 0;

}