**Practical \_\_\_**

**Writeup:**

**Practical \_\_\_**

**Aim:**

**Description:**

**Code:**

**Prac4.pl**

father(joe,paul).

father(joe,mary).

father(joe,hope).

mother(jane,paul).

mother(jane,mary).

mother(jane,hope).

male(paul).

male(joe).

male(raphl).

male(X):-father(X,Y).

female(mary).

female(jane).

female(hope).

female(X):-mother(X,Y).

son\_of(X,Y):- father(Y,X),male(X).

son\_of(X,Y):- mother(Y,X),male(X).

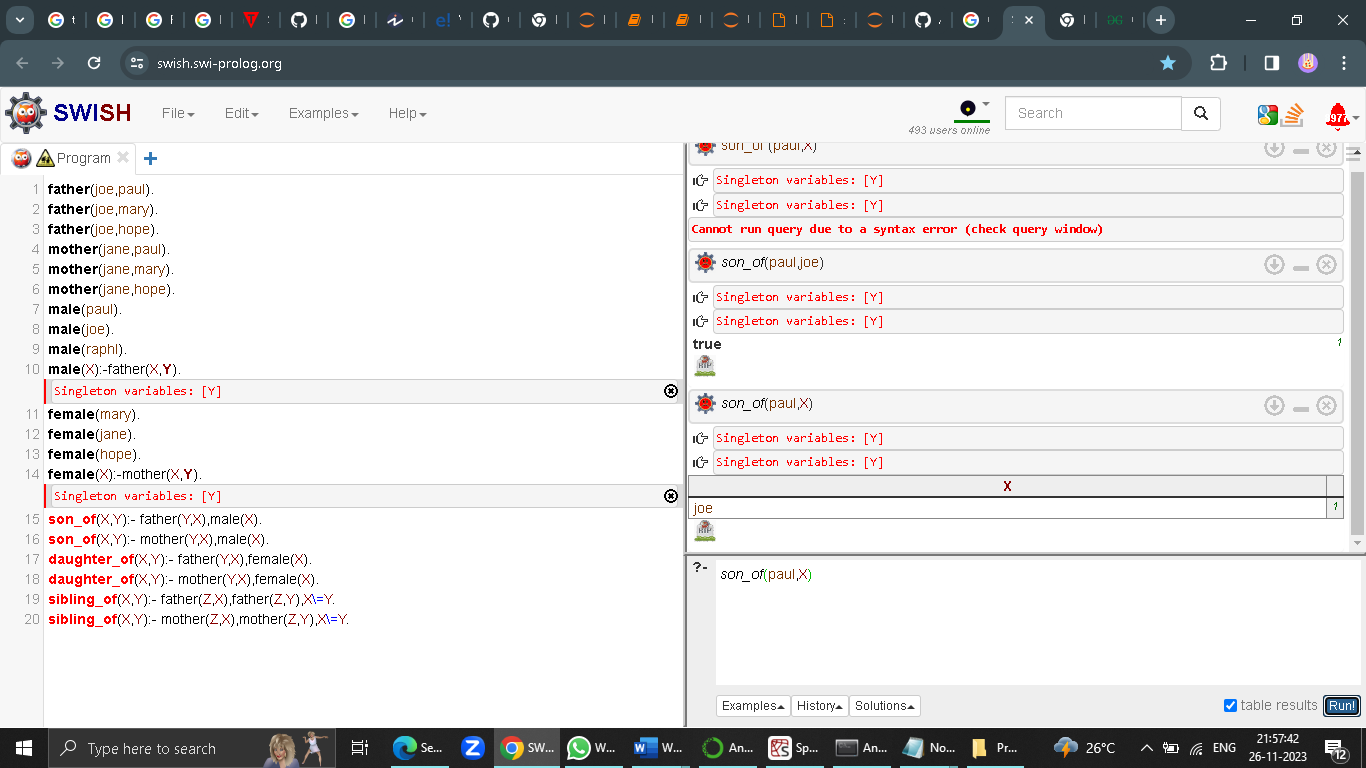
daughter\_of(X,Y):- father(Y,X),female(X).

daughter\_of(X,Y):- mother(Y,X),female(X).

sibling\_of(X,Y):- father(Z,X),father(Z,Y),X\=Y.

sibling\_of(X,Y):- mother(Z,X),mother(Z,Y),X\=Y.

**Output:**



A screenshot of a computer

Description automatically generated

**Practical \_\_\_**

**Writeup:**

**Practical \_\_\_**

**Aim:**

**Description:**

**Code:**

import numpy as np

import skfuzzy as fuzz

import matplotlib.pyplot as plt

from skfuzzy import control as ctrl

from mpl\_toolkits.mplot3d import Axes3D # Required for 3D plotting

# New Antecedent/Consequent objects hold universe variables and membership

# functions

quality = ctrl.Antecedent(np.arange(0, 10, 0.1), 'quality')

service = ctrl.Antecedent(np.arange(0, 10, 0.1), 'service')

tip = ctrl.Consequent(np.arange(0, 25, 0.1), 'tip')

quality['poor'] = fuzz.zmf(quality.universe, 0,5)

quality['average'] = fuzz.gaussmf(quality.universe,5,1)

quality['good'] = fuzz.smf(quality.universe,5,10)

service['poor'] = fuzz.zmf(service.universe, 0,5)

service['average'] = fuzz.gaussmf(service.universe,5,1)

service['good'] = fuzz.smf(service.universe,5,10)

tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])

tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])

tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])

quality['average'].view()

plt.title('Quality')

service['poor'].view()

plt.title('Service')

tip['medium'].view()

plt.title('Tip Medium')

rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])

rule2 = ctrl.Rule(service['average'], tip['medium'])

rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])

rule1.view()

plt.title('Rule 1')

rule2.view()

plt.title('Rule 2')

rule3.view()

plt.title('Rule 3')

tipping\_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])

tipping = ctrl.ControlSystemSimulation(tipping\_ctrl)

tipping.input['quality'] = 6.5

tipping.input['service'] = 9.8

tipping.compute()

print(tipping.output['tip'])

tip.view(sim=tipping)

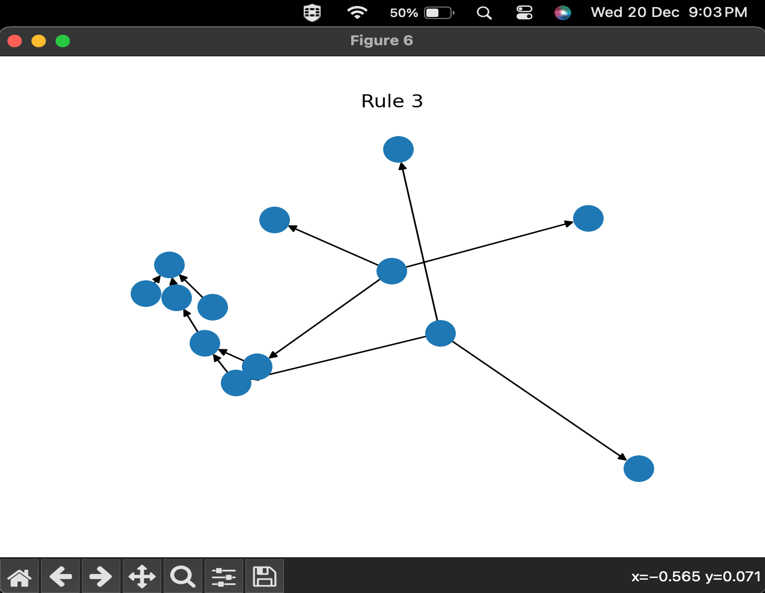
plt.title('Result')

plt.show(block=True)

print('Yash Jayant')

**Output:**

**A screenshot of a computer

Description automatically generated**

A screenshot of a computer

Description automatically generated

A screenshot of a graph

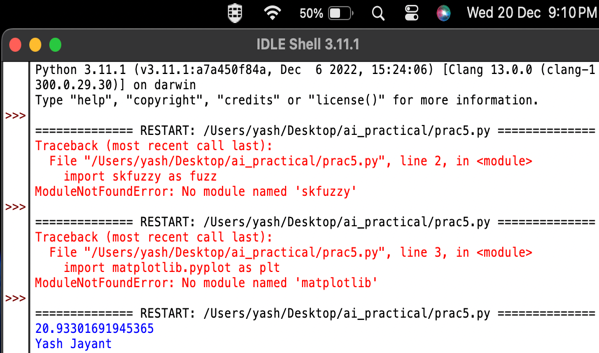
Description automatically generatedA screenshot of a graph

Description automatically generated

A screenshot of a graph

Description automatically generatedA screenshot of a graph

Description automatically generated



**Practical \_\_\_**

**Writeup:**

**Practical \_\_\_**

**Aim:**

**Description:**

**Code:**

**Supervised.py:**

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import confusion\_matrix

import seaborn as sns

from sklearn import datasets

# Importing the dataset (using the iris dataset from sklearn)

iris = datasets.load\_iris()

X = iris.data

y = iris.target

# Splitting the dataset into the training set and test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=0)

# Feature scaling

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

# Fitting logistic regression to the training set

classifier = LogisticRegression(random\_state=0, solver='lbfgs', multi\_class='auto')

classifier.fit(X\_train, y\_train)

# Predicting the test set results

y\_pred = classifier.predict(X\_test)

# Confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

print("Confusion Matrix:")

print(cm)

# Plot confusion matrix using seaborn heatmap

ax = plt.axes()

sns.heatmap(cm, annot=True, annot\_kws={"size": 10}, fmt='d', cmap="Blues", ax=ax)

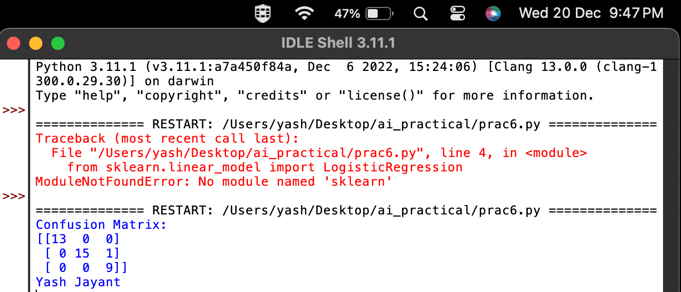
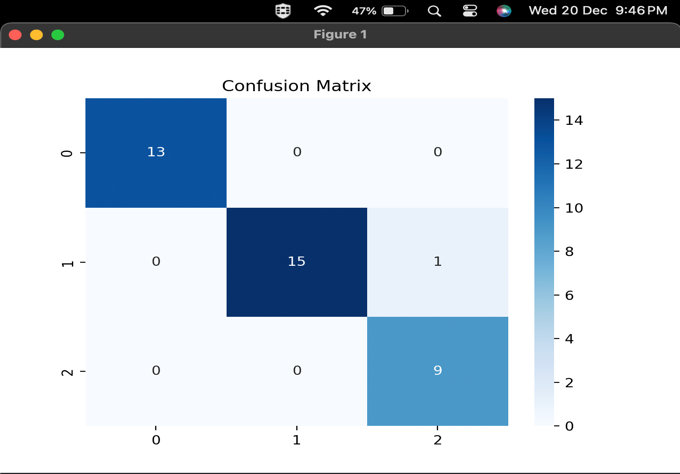
ax.set\_title('Confusion Matrix')

plt.show()

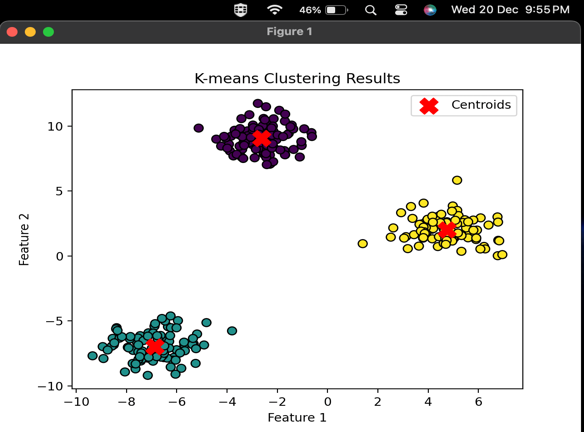
print('Yash Jayant')

**Unsupervised.py**:

**Output:**



A screen shot of a computer screen

Description automatically generated

**Practical \_\_\_**

**Writeup:**

**Practical \_\_\_**

**Aim:**

**Description:**

**Code:**

# Importing Modules

from scipy.cluster.hierarchy import linkage, dendrogram

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.datasets import load\_iris

# Load the Iris dataset

iris = load\_iris()

iris\_df = pd.DataFrame(data=iris.data, columns=iris.feature\_names)

# Extract the target labels (species) for later use

species = iris.target\_names[iris.target]

# Extract the measurements as a NumPy array

samples = iris\_df.values

"""

Perform hierarchical clustering on samples using the

linkage() function with the method='complete' keyword argument.

Assign the result to mergings.

"""

mergings = linkage(samples, method='complete')

"""

Plot a dendrogram using the dendrogram() function on mergings,

specifying the keyword arguments labels=species, leaf\_rotation=90,

and leaf\_font\_size=6.

"""

dendrogram(mergings,

labels=species,

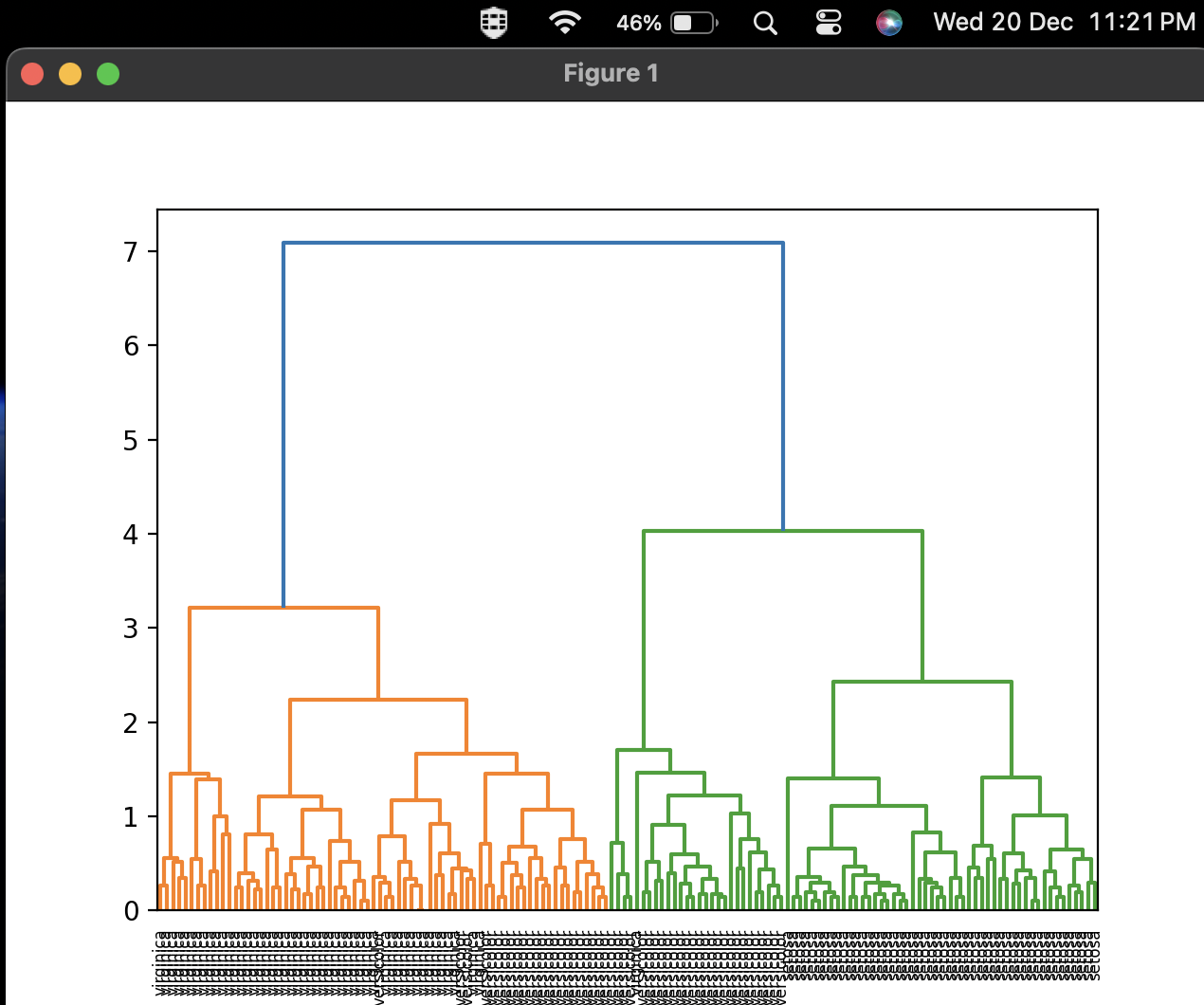
leaf\_rotation=90,

leaf\_font\_size=6,

)

plt.show()

**Output:**



**Practical \_\_\_**

**Writeup:**

**Practical \_\_\_**

**Aim:**

**Description:**

**Code:**

#Import scikit-learn dataset library

from sklearn import datasets

#Import svm model

from sklearn import svm

# Import train\_test\_split function

from sklearn.model\_selection import train\_test\_split

#Import scikit-learn metrics module for accuracy calculation

from sklearn import metrics

#Load dataset

cancer = datasets.load\_breast\_cancer()

# print the names of the 13 features

print("Features: ", cancer.feature\_names)

# print the label type of cancer('malignant' 'benign')

print("Labels: ", cancer.target\_names)

# print data(feature)shape

cancer.data.shape

# print the cancer data features (top 5 records)

print(cancer.data[0:5])

# print the cancer labels (0:malignant, 1:benign)

print(cancer.target)

# Split dataset into training set and test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(cancer.data, cancer.target, test\_size=0.3,random\_state=109) # 70% training and 30% test

#Create a svm Classifier

clf = svm.SVC(kernel='linear') # Linear Kernel

#Train the model using the training sets

clf.fit(X\_train, y\_train)

#Predict the response for test dataset

y\_pred = clf.predict(X\_test)

# Model Accuracy: how often is the classifier correct?

print("Accuracy:",metrics.accuracy\_score(y\_test, y\_pred))

# Model Precision: what percentage of positive tuples are labeled as such?

print("Precision:",metrics.precision\_score(y\_test, y\_pred))

# Model Recall: what percentage of positive tuples are labelled as such?

print("Recall:",metrics.recall\_score(y\_test, y\_pred))

print('Yesh Jayant')

**Output:**

A screenshot of a computer

Description automatically generated



**Practical \_\_\_**

**Writeup:**

**Practical \_\_\_**

**Aim:**

**Description:**

**Code:**

import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float) # two inputs [sleep,study]

y = np.array(([92], [86], [89]), dtype=float) # one output [Expected % in Exams]

X = X / np.amax(X, axis=0) # maximum of X array longitudinally

y = y / 100

# Sigmoid Function

def sigmoid(x):

return 1 / (1 + np.exp(-x))

# Derivative of Sigmoid Function

def derivatives\_sigmoid(x):

return x \* (1 - x)

# Variable initialization

epoch = 5000 # Setting training iterations

lr = 0.1 # Setting learning rate

inputlayer\_neurons = 2 # number of features in data set

hiddenlayer\_neurons = 3 # number of hidden layers neurons

output\_neurons = 1 # number of neurons at output layer

# weight and bias initialization

wh = np.random.uniform(size=(inputlayer\_neurons, hiddenlayer\_neurons)) # weight of the link from input node to hidden node

bh = np.random.uniform(size=(1, hiddenlayer\_neurons)) # bias of the link from input node to hidden node

wout = np.random.uniform(size=(hiddenlayer\_neurons, output\_neurons)) # weight of the link from hidden node to output node

bout = np.random.uniform(size=(1, output\_neurons)) # bias of the link from hidden node to output node

# draws a random range of numbers uniformly of dim x\*y

for i in range(epoch):

# Forward Propogation

hinp1 = np.dot(X, wh)

hinp = hinp1 + bh

hlayer\_act = sigmoid(hinp)

outinp1 = np.dot(hlayer\_act, wout)

outinp = outinp1 + bout

output = sigmoid(outinp)

# Backpropagation

EO = y - output

outgrad = derivatives\_sigmoid(output)

d\_output = EO \* outgrad

EH = d\_output.dot(wout.T)

# how much hidden layer weights contributed to error

hiddengrad = derivatives\_sigmoid(hlayer\_act)

d\_hiddenlayer = EH \* hiddengrad

# dotproduct of nextlayererror and currentlayerop

wout += hlayer\_act.T.dot(d\_output) \* lr

wh += X.T.dot(d\_hiddenlayer) \* lr

print("Input: \n" + str(X))

print("Actual Output: \n" + str(y))

print("Predicted Output: \n", output)

print('Yesh Jayant')

**Output:**



**Practical \_\_\_**

**Writeup:**

**Practical \_\_\_**

**Aim:**

**Description:**

**Code:**

import random

# Number of individuals in each generation

POPULATION\_SIZE = 100

# Valid genes

GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP

QRSTUVWXYZ 1234567890, .-;:\_!"#%&/()=?@${[]}'''

# Target string to be generated

TARGET = "Yash Jayant"

class Individual(object):

'''

Class representing individual in population

'''

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

self.chromosome = chromosome

self.fitness = self.cal\_fitness()

@classmethod

def mutated\_genes(self):

'''

create random genes for mutation

'''

global GENES

gene = random.choice(GENES)

return gene

@classmethod

def create\_gnome(self):

'''

create chromosome or string of genes

'''

global TARGET

gnome\_len = len(TARGET)

return [self.mutated\_genes() for \_ in range(gnome\_len)]

def mate(self, par2):

'''

Perform mating and produce new offspring

'''

# chromosome for offspring

child\_chromosome = []

for gp1, gp2 in zip(self.chromosome, par2.chromosome):

# random probability

prob = random.random()

# if prob is less than 0.45, insert gene

# from parent 1

if prob < 0.45:

child\_chromosome.append(gp1)

# if prob is between 0.45 and 0.90, insert

# gene from parent 2

elif prob < 0.90:

child\_chromosome.append(gp2)

# otherwise insert random gene(mutate),

# for maintaining diversity

else:

child\_chromosome.append(self.mutated\_genes())

# create new Individual(offspring) using

# generated chromosome for offspring

return Individual(child\_chromosome)

def cal\_fitness(self):

'''

Calculate fittness score, it is the number of

characters in string which differ from target

string.

'''

global TARGET

fitness = 0

for gs, gt in zip(self.chromosome, TARGET):

if gs != gt: fitness+= 1

return fitness

# Driver code

def main():

global POPULATION\_SIZE

#current generation

generation = 1

found = False

population = []

# create initial population

for \_ in range(POPULATION\_SIZE):

gnome = Individual.create\_gnome()

population.append(Individual(gnome))

while not found:

# sort the population in increasing order of fitness score

population = sorted(population, key = lambda x:x.fitness)

# if the individual having lowest fitness score ie.

# 0 then we know that we have reached to the target

# and break the loop

if population[0].fitness <= 0:

found = True

break

# Otherwise generate new offsprings for new generation

new\_generation = []

# Perform Elitism, that mean 10% of fittest population

# goes to the next generation

s = int((10\*POPULATION\_SIZE)/100)

new\_generation.extend(population[:s])

# From 50% of fittest population, Individuals

# will mate to produce offspring

s = int((90\*POPULATION\_SIZE)/100)

for \_ in range(s):

parent1 = random.choice(population[:50])

parent2 = random.choice(population[:50])

child = parent1.mate(parent2)

new\_generation.append(child)

population = new\_generation

print("Generation: {}\tString: {}\tFitness: {}".\

format(generation,

"".join(population[0].chromosome),

population[0].fitness))

generation += 1

print("Generation: {}\tString: {}\tFitness: {}".\

format(generation,

"".join(population[0].chromosome),

population[0].fitness))

if \_\_name\_\_ == '\_\_main\_\_':

main()

**Output:**



**Practical \_\_\_**

**Writeup:**

**Practical \_\_\_**

**Aim:**

**Description:**

**Code:**

import numpy as np

class QLearningAgent:

def \_\_init\_\_(self, num\_states, num\_actions, learning\_rate=0.1, discount\_factor=0.9, exploration\_rate=0.1):

# Initialize Q-table with zeros

self.q\_table = np.zeros((num\_states, num\_actions))

self.learning\_rate = learning\_rate

self.discount\_factor = discount\_factor

self.exploration\_rate = exploration\_rate

def select\_action(self, state):

# Choose the action with the highest Q-value, with exploration

return np.argmax(self.q\_table[state, :]) if np.random.rand() > self.exploration\_rate else np.random.choice(len(self.q\_table[state, :]))

def update\_q\_table(self, state, action, reward, next\_state):

# Update Q-value using the Q-learning update rule

best\_next\_action = np.argmax(self.q\_table[next\_state, :])

self.q\_table[state, action] += self.learning\_rate \* (reward + self.discount\_factor \* self.q\_table[next\_state, best\_next\_action] - self.q\_table[state, action])

class ParkingEnvironment:

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

# Initialize parking environment with 3 states and 2 actions

self.num\_states = 3

self.num\_actions = 2

self.goal\_state = 2

self.agent\_state = 0

self.done = False

def reset(self):

# Reset the environment for a new episode

self.agent\_state = 0

self.done = False

def step(self, action):

if self.done:

return self.agent\_state, 0, self.done

# Update agent's position based on the action

if action == 0: # Move left

self.agent\_state = max(0, self.agent\_state - 1)

else: # Move right

self.agent\_state = min(self.num\_states - 1, self.agent\_state + 1)

# Provide reward based on the agent reaching the goal state

reward = 1 if self.agent\_state == self.goal\_state else 0

self.done = (self.agent\_state == self.goal\_state)

return self.agent\_state, reward, self.done

# Main loop

num\_states = 3

num\_actions = 2

num\_episodes = 100

# Create Q-learning agent and parking environment

agent = QLearningAgent(num\_states, num\_actions)

environment = ParkingEnvironment()

# Training loop

for episode in range(num\_episodes):

environment.reset()

# Episode loop

while not environment.done:

state = environment.agent\_state

action = agent.select\_action(state)

next\_state, reward, done = environment.step(action)

agent.update\_q\_table(state, action, reward, next\_state)

# Test the trained agent

test\_episodes = 5

for episode in range(test\_episodes):

environment.reset()

# Episode loop for testing

while not environment.done:

state = environment.agent\_state

action = agent.select\_action(state)

next\_state, reward, done = environment.step(action)

print(f"Test Episode {episode + 1}, Total Reward: {reward}")

**Output:**

A screenshot of a phone

Description automatically generated

**Practical \_\_\_**

**Writeup:**

**Practical \_\_\_**

**Aim:**

**Description:**

**Code:**

# Importing the required libraries

import nltk

from nltk import CFG

# Defining the grammar rules

grammar = CFG.fromstring("""

S -> NP VP

NP -> Det N | Det N PP

VP -> V NP | V NP PP

PP -> P NP

Det -> 'The' | 'a' | 'the'

N -> 'dog' | 'cat' | 'house' | 'car'

V -> 'chased' | 'ate' | 'drove'

P -> 'in' | 'on' | 'at'

""")

# Creating the parser

parser = nltk.ChartParser(grammar)

# Parsing a sentence

sentence = "The dog chased the cat"

for tree in parser.parse(sentence.split()):print(tree)

print('Yesh Jayant')

**Output:**

A screenshot of a phone

Description automatically generated

**Practical \_\_\_**

**Writeup:**

**Practical \_\_\_**

**Aim:**

**Description:**

**Code:**

from rdflib import Graph, Literal, URIRef

from rdflib.namespace import FOAF, RDF

# Create an RDF graph

g = Graph()

# Use a fictional domain

base\_uri = URIRef("http://mysemanticsimulation.org/")

# Add some triples to the graph

g.add((base\_uri, RDF.type, FOAF.Person))

g.add((base\_uri, FOAF.name, Literal("John Doe")))

g.add((base\_uri, FOAF.age, Literal(25)))

# Save the graph to a file

g.serialize(destination='example.rdf', format='xml')

# Query the graph using SPARQL

query = """

SELECT ?name ?age

WHERE {

?person rdf:type foaf:Person .

?person foaf:name ?name .

?person foaf:age ?age .

}

"""

result = g.query(query, initBindings={'person': base\_uri})

# Print query results

for row in result:

print(f"Name: {row['name']}, Age: {row['age']}")

print("Yesh Jayant")

**Output:**

