

University of Chittagong
Computer Science and Engineering

Lab report on Machine Learning lab

Course: Machine Learning Lab (CSE 816)

Name: Lab report

Submitted to:

Dr. Mohammad Osiur Rahman

Professor, Dept. of Computer Science & Engineering

University of Chittagong, Bangladesh.

Submitted by:

Ishra Naznin

ID: 18701069

Session: 2017-18

Dept. of Computer Science & Engineering

University of Chittagong, Bangladesh

1. Write a program to implement a perceptron network for recognizing orange (p1) or an apple (p2) patterns. Test the operation of your networks by applying several different input patterns.

Ans: An artificial neuron termed a perceptron can be used to carry out basic operations like classification and regression. It has just one layer of neurons since it is a single-layer neural network. In a perceptron, each neuron has a number of inputs, each of which has a weight. The inputs are multiplied by their respective weights before being added together, occasionally with bias. The output of the neuron is then determined by passing this sum through an activation function.

Implementation of program: To implement a perceptron network for recognizing orange (p1) or an apple (p2) patterns, a program of the Perceptron algorithm in Python with test operations is given below. The perceptron, a basic form of artificial neural network that may be used to categorize data, is implemented in this code. For a binary classification problem this code first specifies the input patterns and related labels. Following initialization, the perceptron network is trained using the input patterns by the code. The network may be used to predict the class of fresh input patterns once it has been trained. A straightforward GUI is also included in the code, which enables users to insert fresh input arrays and view the network's prediction output.

```
import numpy as np
class Perceptron:
    def __init__(self, inputs, labels):
        self.inputs = inputs
        self.labels = labels
        self.weights = np.random.uniform(-1, 1, (3,))
        self.bias = np.random.uniform(-1, 1)
    def activation_function(self, x):
        if x >= 0:
            return 1
        else:
            return 0
    def train(self, epochs):
        for epoch in range(epochs):
            for i in range(len(self.inputs)):
                prediction = self.predict(self.inputs[i])
                error = self.labels[i] - prediction
                self.weights += error * self.inputs[i]
                print(self.weights)
            self.bias += error
```

```

def predict(self, inputs):
    dot_product = np.dot(inputs, self.weights) + self.bias
    prediction = self.activation_function(dot_product)
    return prediction
# Define input patterns
inputs = np.array([[1, 1, -1], [1, -1, -1]])
# Define corresponding labels (1 for apples and -1 for oranges)
labels = np.array([1,-1])
# Initialize the perceptron network
perceptron = Perceptron(inputs, labels)
# Train the network
perceptron.train(100)
# Test the network with some inputs
test_inputs = np.array([[1, 1, 1], [1, -1, 1], [1, -1, -1]])
for inputs in test_inputs:
    prediction = perceptron.predict(inputs)
    if prediction==1:
        print(f'Input:{inputs} Prediction: Apple")
    else:
        print(f'Input:{inputs} Prediction: Orange")
import tkinter as tk
# Define a function to handle the button click event

```

```

def button_click():
    # Get the input array from the entry widget
    input_str = input_entry.get()
    input_arr = [int(x.strip()) for x in input_str.split(",")]

    # Display the input array in the label widget
    output_label.config(text=f'Input array: {input_arr}')
    prediction = perceptron.predict(input_arr)
    if prediction==1:
        print(f'Input: {input_arr}, Prediction: Apple")
        output_label.config(text=f'Output: Prediction: Apple")
    else:
        print(f'Input: {input_arr}, Prediction: Orange")
        output_label.config(text=f'Output: Prediction: Orange")
# Create the GUI
root = tk.Tk()
root.title('Array Input')

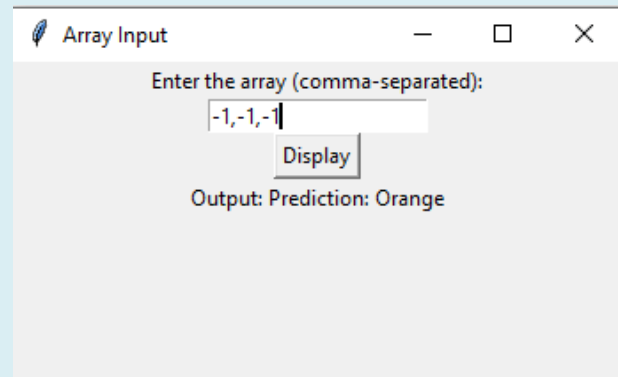
# Create a label for the input entry widget
input_label = tk.Label(root, text='Enter the array (comma-separated):')
input_label.pack()

```

Output:

```
Input: [-1, -1, -1], Prediction: Orange  
Input: [-1, -1, -1], Prediction: Orange  
Input: [-1, -1, -1], Prediction: Orange
```

Fig1: Test operation output of apple-orange pattern recognition problem



Array Input

Enter the array (comma-separated):

Output: Prediction: Orange

Fig1: User Interface of apple-orange pattern recognition problem

2. Write a program to implement a Hamming / Hopfield network for recognizing banana (p1) or a pineapple (p2) patterns. Test the operation of your networks by applying several different input patterns.

Ans: Recurrent neural networks can be used to store and remember patterns. Examples include Hamming and Hopfield networks. A fully connected network having a binary state is referred to as a Hamming network. There are actually two possible states for each neuron in the network: +1 or -1. When an incorrect version of a pattern is provided to the network after it has been trained with a collection of patterns, the weights of the connections between the neurons are changed to ensure that the network will still pick up the right pattern. Although a Hopfield network might have either a binary or continuous state, it is still a fully linked network. When a collection of patterns are shown to the network during training, the weights of the connections between the neurons are changed to ensure that the network will always converge to one of the stored patterns when a new pattern is provided.

Implementation of program: To implement a Hamming network for recognizing banana (p1) or a pineapple (p2) patterns a python program given with several test operation:

```
import numpy as np  
# Initialize weights and biases  
weights = np.array([[-1, 1, -1], [-1, -1, 1]])  
biases = np.array([3, 3])
```

```

# Define activation function
def activation(x):
    if x >= 0:
        return x
    else:
        return 0
def activationForward(x):
    return x
weigh2=np.array([[1,-0.5],[-0.5,1]])
# Define the forward pass
def forward(inputs, weights, biases):
    output = np.dot(weights,inputs) + biases
    return [activationForward(x) for x in output]
def recurrent(outputs, weigh2):
    output2 = np.dot(weigh2, outputs)
    return [activation(x) for x in output2]
# Define the fruit classification function
def classify_fruit(inputs):
    outputs = forward(inputs, weights, biases)
    outputs2=recurrent(outputs, weigh2)
    if outputs2[0] > outputs2[1]:
        return "Banana"
    else:

```

```

        return "Pineapple"

# Define input vectors for fruit
test_input = np.array([1,- 1, 1])
print(f"test {test_input}, {classify_fruit(test_input)}")
# Test the operation of the network
import tkinter as tk

# Define a function to handle the button click event
def button_click():
    # Get the input array from the entry widget
    input_str = input_entry.get()
    input_arr = [int(x.strip()) for x in input_str.split(",")]

    # Display the input array in the label widget
    output_label.config(text=f"Input: {input_arr}, Output: {classify_fruit(
input_arr)}")

# Create the GUI
root = tk.Tk()

```

```

root.title('Array Input')

# Create a label for the input entry widget
input_label = tk.Label(root, text='Enter the array (comma-separated):')
input_label.pack()

# Create an entry widget for the input array
input_entry = tk.Entry(root)
input_entry.pack()

# Create a button to display the input array
display_button = tk.Button(root, text='Display', command=button_click)
display_button.pack()

# Create a label for the output
output_label = tk.Label(root, text='')
output_label.pack()

root.mainloop()

```

Output:

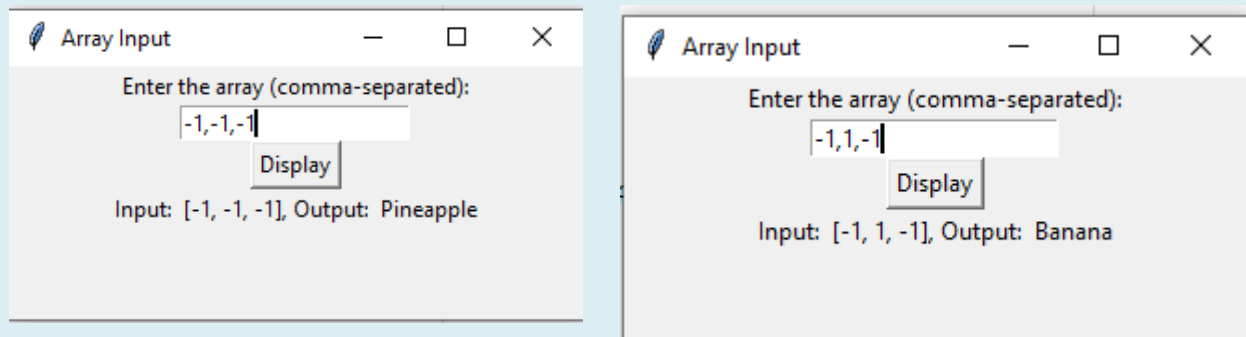


Fig2: User Interface of Hamming network to recognize banana-pineapple pattern recognition problem

3. Write a program for training the neural network and then Test the operation of your networks by applying several different input patterns.

Ans: A machine learning approach with neural networks is modeled after the human brain. It is a network of interconnected nodes, each of which performs a simple computation. The weights of the connections between the nodes are changed once the network has been trained with a set of data so that it can successfully predict the output for any input.

Implementation of program: An implementation of a program for training the neural network and then Test the operation of the networks by applying several different input patterns in python is given below: here we assume the learning rate is 0.1 and the initial weight vector is $w_0 = (1, -1, 0)^T$. The perceptron learning rule is implemented in this code for the specified training set. The training set, a collection of input patterns and labels, is initially defined by the code. Next, the program initializes the weight vector, a vector of weights used to categorize fresh input patterns. The maximum number of iterations and learning rate are then determined by the programming. The learning rate is a parameter that determines the amount by which the weights are modified following in each iteration. The maximum number of iterations is the total number of training cycles for the perceptron. Utilizing the perceptron learning rule, the code then trains the perceptron. The perceptron learning rule is an iterative process that alters the perceptron's weights until the perceptron can properly categorize each input pattern in the training set. The code runs tests on the perceptron using fresh input patterns once it has been trained. Following that, the code shows the perceptron's predictions for the fresh input patterns. Users may add fresh input arrays and view the perceptron's predictions using a straightforward GUI that is also included in the code.

```
import numpy as np
# define the training set
X = np.array([[1, 0, 1], [0, -1, -1], [-1, -0.5, -1]])
y = np.array([-1, 1, 1])
# initialize the weight vector
w = np.array([1, -1, 0])
bias = 0.1
max_iterations = 100 # set the learning rate and maximum number of iterations

# train the neural network using the perceptron learning rule
for i in range(max_iterations):
    for j in range(len(X)):
        x = X[j]
```

```

    y_hat = np.sign(np.dot(w, x))
    error = y[j] - y_hat
    w = w+ error * bias * x
    bias = bias+ bias * error
# check for convergence
if np.all(np.sign(np.dot(X, w)) == y):
    print(f"Converged after {i+1} iterations.")
    break
else:
    print(f"Did not converge after {max_iterations} iterations.")

print("Final weight vector: ", w)
print("Final bias : ",bias)
# test the network with new input patterns
x_test = np.array([[1, 1, 1], [-1, 0, 1], [0, 1, -1]])
y_test = np.sign(np.dot(x_test, w))
print("Predictions for test set: ", y_test)
import tkinter as tk

# Define a function to handle the button click event
def button_click():
    # Get the input array from the entry widget
    input_str = input_entry.get()

    input_list = input_str.split(',')
    input_arr = np.array([list(map(int, row.split(','))) for row in input_list])
    # Display the input array in the label widget
    y_test = np.sign(np.dot(input_arr, w))
    output_label.config(text=f"Predictions for test set: { y_test } \n weight : { w}\n bias: {bias}")

# Create the GUI
root = tk.Tk()
root.title('Array Input')

# Create a label for the input entry widget
input_label = tk.Label(root, text='Enter the array (comma-separated):')
input_label.pack()

# Create an entry widget for the input array
input_entry = tk.Entry(root)
input_entry.pack()

# Create a button to display the input array

```



```

display_button = tk.Button(root, text='Display', command=button_click)
display_button.pack()

# Create a label for the output
output_label = tk.Label(root, text='')
output_label.pack()

root.mainloop()

```

Output:

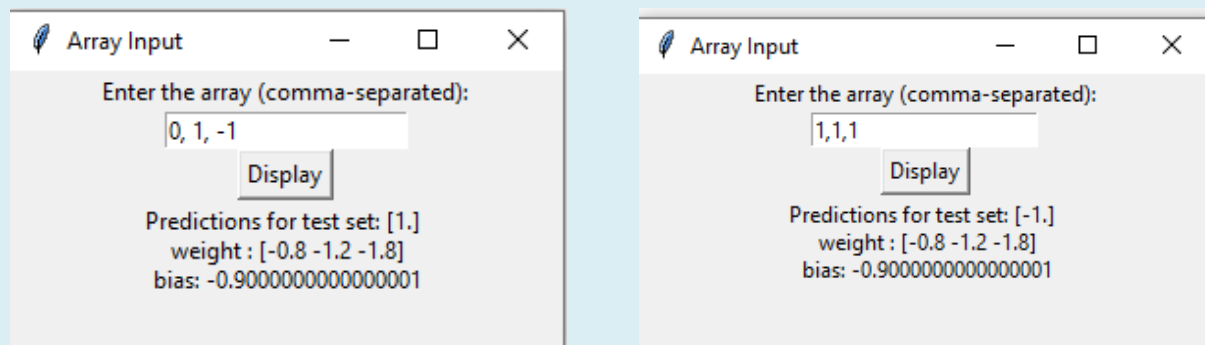


Fig3: User Interface of the Python program that implements the perceptron learning rule for the given training set:

4. Write a program to solve the following classification problem with the perceptron rule. Apply each input vector in order, for as many repetitions as it takes to ensure that the problem is solved. Draw a graph of the problem only after you have found a solution.

$$\left\{ p_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, t_1 = 0 \right\} \left\{ p_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, t_2 = 1 \right\} \left\{ p_3 = \begin{bmatrix} -2 \\ 2 \end{bmatrix}, t_3 = 0 \right\} \left\{ p_4 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t_4 = 1 \right\}$$

$$W(0) = \begin{bmatrix} 0 & 0 \end{bmatrix} \quad b(0) = 0.$$

Ans: The implementation in python of the problem is given below:

```
import numpy as np
import matplotlib.pyplot as plt
# Define the input vectors and their desired output values
p1 = np.array([2, 2])
t1 = 0
p2 = np.array([1, -2])
t2 = 1
p3 = np.array([-2, 2])
t3 = 0
p4 = np.array([-1, 1])
t4 = 1
# Initialize the weight vector and bias
W = np.array([0, 0])
b = 0
# Define the activation function
def activation(x):
    return 1 if x >= 0 else 0

# Train the neural network using the perceptron rule
max_iterations = 1000
for i in range(max_iterations):
    # Apply each input vector in order
    for p, t in [(p1, t1), (p2, t2), (p3, t3), (p4, t4)]:
        # Calculate the weighted sum of inputs
        a = np.dot(W, p) + b
        # Calculate the output of the neuron
        y = activation(a)
        # Update the weights and bias if the output is incorrect
        if y != t:
            W = W + (t - y) * p
            b = b + (t - y)

    # Check if the problem is solved
    if (np.dot(W, p1) + b >= 0 and np.dot(W, p3) + b < 0) or \
       (np.dot(W, p1) + b < 0 and np.dot(W, p3) + b >= 0) or \
       (np.dot(W, p2) + b >= 0 and np.dot(W, p4) + b < 0) or \
       (np.dot(W, p2) + b < 0 and np.dot(W, p4) + b >= 0):
        break

# Print the solution
print("The problem is solved after", i+1, "iterations.")
```

```

print("The weight vector is", W)
print("The bias is", b)
# Plot the input vectors and the decision boundary
plt.scatter(p1[0], p1[1], c='g', marker='o')
plt.scatter(p2[0], p2[1], c='b', marker='o')
plt.scatter(p3[0], p3[1], c='r', marker='s')
plt.scatter(p4[0], p4[1], c='black', marker='s')
# Add text labels
plt.text(p1[0]+0.1, p1[1], "P1", color='g')
plt.text(p2[0]+0.1, p2[1], "P2", color='b')
plt.text(p3[0]+0.1, p3[1], "P3", color='r')
plt.text(p4[0]+0.1, p4[1], "P4", color='black')
x = np.linspace(-3, 3)
y = (-b - W[0]*x) / W[1]
plt.plot(x, y)
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Perceptron Classification')
plt.show()
# Create the GUI
import tkinter as tk
root = tk.Tk()
root.title('4. Perceptron:')

```

```

# Define a function to handle the button click event
def button_click():
    output_label.config(text=f"The problem is solved after {i+1} iterations.\n The
weight vector is {W} \n The bias is {b}")
display_button = tk.Button(root, text='Display', command=button_click)
display_button.pack()

# Create a label for the output
output_label = tk.Label(root, text='')
output_label.pack()

root.mainloop()

```

Output:

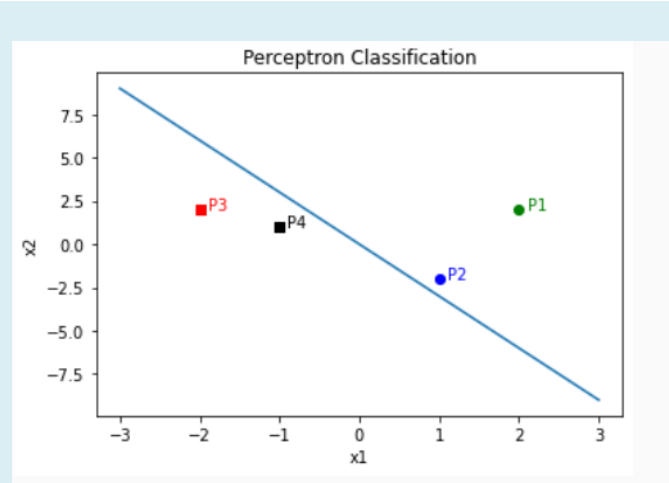


Fig 4.1: Graph of the problem after found a solution

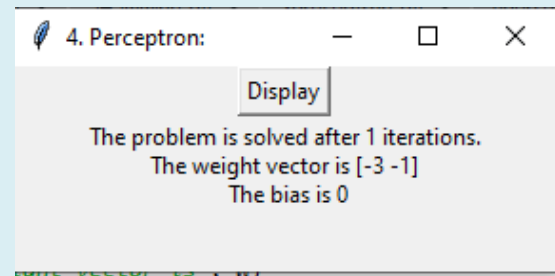


Fig 4.2: User Interface output of the Python program that implements the perceptron learning rule for the given training set

5. Consider the problem of maximizing the function:

$$f(x) = \frac{2x - x^2}{16}$$

Here, x is allowed to vary between 0 and 31. Write a program to solve this using a genetic algorithm. Note that the relative fitness function $g(x)$ is defined as below:

$$g(x) = \frac{f(x)}{\sum_{x \in p(k)} f(x)}$$

Ans: A member of the wider family of evolutionary algorithms (EA), a genetic algorithm (GA) is a search heuristic that takes its cues from the phenomenon of natural selection. Genetic algorithms frequently employ biologically inspired operators including mutation, crossover, and selection to produce high-quality solutions to optimization and search issues. In a genetic algorithm, a population of potential solutions to an optimization problem (referred to as people, animals, organisms, or phenotypes) evolves toward superior answers. Traditionally, solutions are represented in binary as strings of 0s and 1s, although alternative encodings are also feasible. Each potential solution has a set of attributes (its chromosomes or genotype) that may be changed and modified. A generation is the term used to describe the population in each iteration of the evolution, which typically begins with a population of randomly produced individuals.

Implementation of the problem: A python implementation of the given problem to solve using genetic algorithm is given below. The evolutionary approach for optimizing functions is implemented in this code. The parameters of the genetic algorithm are initially defined in the code, along with the function that has to be optimized and the search window. Next, the code specifies the population, the fitness function, the selection operator, the crossover operator, the mutation operator, and the initialization and representation of each individual. After that, the function iterates over the generations after initializing the population. Each generation begins with an assessment of the population's fitness, followed by the selection of the parents for reproduction, the creation of the offspring, their mutation, and the replacement of the least fit people with the offspring. Following that, the code outputs the top outcome.

```
import random
import math
import tkinter as tk
# Define a function to handle the button click event
def button_click():
    # Get the function expression from the entry widget
    function_expression = input_entry.get()
    # define the function to be optimized
    def function_to_optimize(x):
        return ((2*x)-(x**2))/16

    # define the search interval
    search_interval = (0, 31)
    # define the genetic algorithm parameters
    population_size = 10
    mutation_rate = 0.1
    generations = int(generations_entry.get()) # Get the number of generations from
the entry widget

    # define the individual representation and initialization
    def create_individual():
        return random.uniform(search_interval[0], search_interval[1])

# define the fitness function
def fitness(individual):
    return function_to_optimize(individual)
# define the selection operator
def selection(population):
    return random.sample(population, 2)
```

```
# define the crossover operator
```

```
def crossover(parents):
```

```
    return (parents[0] + parents[1]) / 2
```

```
# define the mutation operator
```

```
def mutation(individual):
```

```
    return individual + random.uniform(-1, 1) * mutation_rate
```

```
# initialize the population
```

```
population = [create_individual() for i in range(population_size)]
```

```
# iterate over the generations
```

```
for i in range(generations):
```

```
    # evaluate the fitness of the population
```

```
    fitness_values = [fitness(individual) for individual in population]
```

```
    # select the parents for reproduction
```

```
    parents = [selection(population) for j in range(population_size // 2)]
```

```
    # perform crossover to generate the offspring
```

```
    offspring = [crossover(parents[j]) for j in range(population_size // 2)]
```

```
    # perform mutation on the offspring
```

```
    offspring = [mutation(individual) for individual in offspring]
```

```
    # replace the least fit individuals with the offspring
```

```
    print([type(p) for p in population])
```

```
    print([type(f) for f in fitness_values])
```

```
    population = sorted(list(zip(population, fitness_values)), key=lambda x: -  
x[1])[:population_size // 2]
```

```
    population = [individual for individual, fitness in population] + offspring
```

```
# print the best solution found
```

```
best_individual = max(population, key=fitness)
```

```
print(f"The maximum value of the function is : { fitness(best_individual)} \n at x  
= {best_individual}")
```

```
output_label.config(text=f"The maximum value of the function is:\n  
{fitness(best_individual)}\n at x = { best_individual}")
```

```
# Create the GUI
```

```
root = tk.Tk()
```

```
root.title('Function Input')
```

```
# Create a label for the input entry widget
```

```
input_label = tk.Label(root, text='Enter a function (use x as variable):')
```

```
input_label.pack()
```

```
# Create an entry widget for the function input
```

```

input_entry = tk.Entry(root)
input_entry.pack()

# Create a label for the generations entry widget
generations_label = tk.Label(root, text='Enter the number of generations:')
generations_label.pack()

# Create an entry widget for the number of generations
generations_entry = tk.Entry(root)
generations_entry.pack()

# Create a button to display the validity of the function
display_button = tk.Button(root, text='Test Function', command=button_click)
display_button.pack()

# Create a label for the output
output_label = tk.Label(root, text='')
output_label.pack()

root.mainloop()

```

Output:

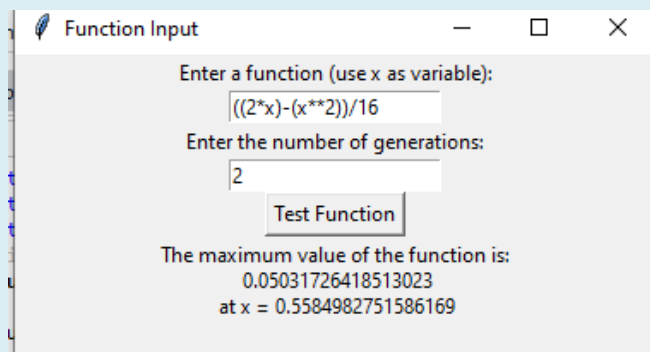


Fig 5.1: User interface of the solution of given function at Generation is 2.

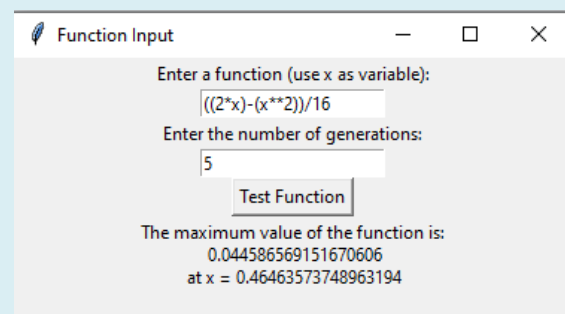


Fig 5.2: User interface of the solution of given function at Generation is 5.

6. Consider the problem of maximizing the function

$$f(x) = \frac{-x^2}{10} + 3x$$

x is allowed to vary [0 - 31]. Write a program to solve this using a genetic algorithm.

Ans: The implementation to solve the function using genetic algorithm is given below:

```
import random
import math
import tkinter as tk
# Define a function to handle the button click event
def button_click():
    # Get the function expression from the entry widget
    function_expression = input_entry.get()

    # define the function to be optimized
    def function_to_optimize(x):
        return ((-x**2)/10)+(3*x)
```

```
# define the search interval
search_interval = (0, 31)
# define the genetic algorithm parameters
population_size = 10
mutation_rate = 0.1
generations = int(generations_entry.get())

# define the individual representation and initialization
def create_individual():
    return random.uniform(search_interval[0], search_interval[1])

# define the fitness function
def fitness(individual):
    return function_to_optimize(individual)

# define the selection operator
def selection(population):
    return random.sample(population, 2)

# define the crossover operator
def crossover(parents):
    return (parents[0] + parents[1]) / 2
```



```

# define the mutation operator
def mutation(individual):
    return individual + random.uniform(-1, 1) * mutation_rate

# initialize the population
population = [create_individual() for i in range(population_size)]

# iterate over the generations
for i in range(generations):
    # evaluate the fitness of the population
    fitness_values = [fitness(individual) for individual in population]
    # select the parents for reproduction
    parents = [selection(population) for j in range(population_size // 2)]
    # perform crossover to generate the offspring
    offspring = [crossover(parents[j]) for j in range(population_size // 2)]
    # perform mutation on the offspring
    offspring = [mutation(individual) for individual in offspring]
    # replace the least fit individuals with the offspring
    print([type(p) for p in population])

```

```

print([type(f) for f in fitness_values])
population = sorted(list(zip(population, fitness_values)), key=lambda x: -
x[1][:population_size // 2])
population = [individual for individual, fitness in population] + offspring

# print the best solution found
best_individual = max(population, key=fitness)
print(f"The maximum value of the function is : { fitness(best_individual)} \n at x
= {best_individual}")
output_label.config(text=f"The maximum value of the function is:\n
{fitness(best_individual)}\n at x = { best_individual}")
# Create the GUI
root = tk.Tk()
root.title('Function Input')

# Create a label for the input entry widget
input_label = tk.Label(root, text='Enter a function (use x as variable):')
input_label.pack()

# Create an entry widget for the function input
input_entry = tk.Entry(root)
input_entry.pack()

```

```

# Create a label for the generations entry widget
generations_label = tk.Label(root, text='Enter the number of generations:')
generations_label.pack()

# Create an entry widget for the number of generations
generations_entry = tk.Entry(root)
generations_entry.pack()

# Create a button to display the validity of the function
display_button = tk.Button(root, text='Test Function', command=button_click)
display_button.pack()

# Create a label for the output
output_label = tk.Label(root, text='')
output_label.pack()

root.mainloop()

```

Output:

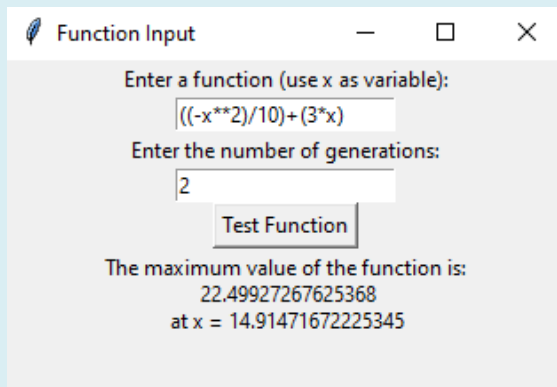


Fig 6.1: User interface of the solution of given function at Generation is 2.

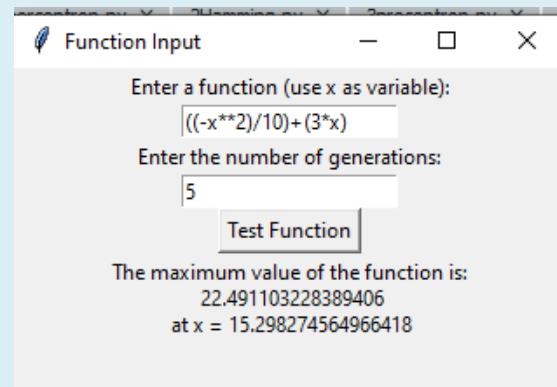


Fig 6.2: User interface of the solution of given function at Generation is 5.

7. Write a program to implement K-means clustering algorithm. Consider the data set consisting of the scores of two variables on each of seven individuals. Test the program using above data set where $k = 2$, i.e., this data set is to be grouped into two clusters.

Ans: An unsupervised learning approach called K-means clustering divides data points into k clusters, where k is a user-defined integer. The algorithm operates by updating the cluster centroids after iteratively assigning data points to clusters until the process converges.

Implementation of the program: A python program to implement the given problem is given below:

```
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
dataset = pd.read_csv('data 1.csv')
x1 = np.array(dataset['A'])
x2 = np.array(dataset['B'])
x3 = np.array(dataset['Subject'])

X = np.array(list(zip(x1, x2)))
colors = ['b', 'g', 'r']
```

```
markers = ['o', 'v', 's']
plt.ylabel('Variable B')

kmeans = KMeans(n_clusters=2).fit(X)
plt.scatter(kmeans.cluster_centers_[0, 0],
            kmeans.cluster_centers_[0, 1],
            s = 200, c = 'red', label = 'Centroids')
for i, l in enumerate(kmeans.labels_):
    plt.plot(x1[i], x2[i], color=colors[l], marker=markers[l])
    plt.text(x1[i]+0.1, x2[i]+0.1, f'{x3[i]}')
plt.xlabel('Variable A')
plt.legend()
plt.show()
```

Output:

Subject	A	B
1	1	1
2	1.5	2
3	3	4
4	5	7
5	3.5	5
6	4.5	5
7	3.5	4.5

Fig 7.1: Test data saved in a .csv file

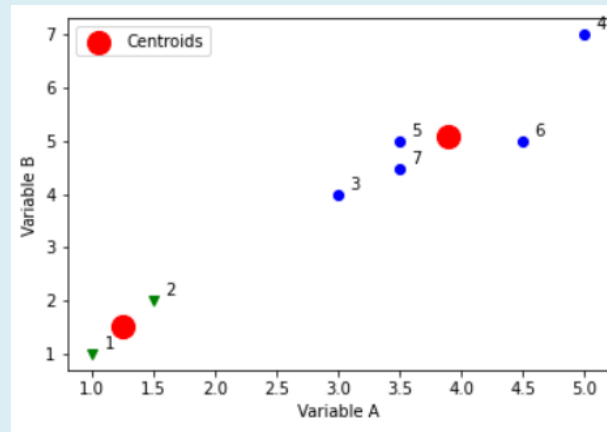


Fig 7.2: The Plot diagram of output clusters, k=2

8. Suppose that the machine learning task is to cluster the following students CGPA into three clusters, i.e., $k = 3$:

<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>	<u>F</u>
3.45	3.78	2.98	3.24	4.0	3.9

The distance function is Euclidean distance. Suppose initially we assign A, C, E as the center of each cluster. Write a program using k-means algorithm to show only the final three clusters.

Ans:

Implementation of the program: A python program to implement the given problem is given below:

```

import numpy as np
import tkinter as tk
# Create the Tkinter window
window = tk.Tk()
window.title("K-Means Clustering")
window.geometry("400x300")
# Create input fields for number of clusters, cluster center points, and test number
cluster_label = tk.Label(window, text="Number of Clusters:")
cluster_label.pack()
cluster_entry = tk.Entry(window)
cluster_entry.pack()
centers_label = tk.Label(window, text="Cluster Center Points (comma-separated):")
centers_label.pack()
centers_entry = tk.Entry(window)
centers_entry.pack()
test_label = tk.Label(window, text="Test Number:")
test_label.pack()
test_entry = tk.Entry(window)
test_entry.pack()
# Function to run the clustering algorithm and display results

```

```

# Function to run the clustering algorithm and display results
def run_clustering():
    # Get input values from the user
    k = int(cluster_entry.get())
    centroids = np.array([float(x) for x in centers_entry.get().split(",")])
    test_point = float(test_entry.get())

    # Student CGPA
    data = np.array([3.45, 3.78, 2.98, 3.24, 4.0, 3.9])

    def euclidean_distance(x1, x2):
        return np.sqrt(np.sum((x1 - x2) ** 2))

    # Define a function to assign each point to the nearest center
    def assign_to_clusters(points, centers):
        distances = np.abs(points[:, np.newaxis] - centers)
        cluster_assignments = np.argmin(distances, axis=1)
        return cluster_assignments

```

```

def find_clusters(data, centroids):
    clusters = np.zeros(len(data))
    for i in range(len(data)):
        dist = np.zeros(k)
        for j in range(k):
            dist[j] = euclidean_distance(data[i], centroids[j])
        cluster = np.argmin(dist)
        clusters[i] = cluster
    return clusters

def update_centroids(data, clusters, centroids):
    for i in range(k):
        points = [data[j] for j in range(len(data)) if clusters[j] == i]
        centroids[i] = np.mean(points, axis=0)
# Repeat until convergence
while True:
    clusters = find_clusters(data, centroids)
    prev_centroids = centroids
    update_centroids(data, clusters, centroids)
    if np.allclose(prev_centroids, centroids):
        break
# Display the final clusters
result_label = tk.Label(window, text="Final Clusters:")
result_label.pack()

```

```

for i in range(k):
    points = [data[j] for j in range(len(data)) if clusters[j] == i]
    cluster_result = f"Cluster {i + 1}: {points}"
    cluster_result_label = tk.Label(window, text=cluster_result)
    cluster_result_label.pack()

# Assign the test number to a cluster
test_assignment = assign_to_clusters(np.array([test_point]), centroids)
test_result = f"Test point {test_point} belongs to cluster {test_assignment + 1}"
test_result_label = tk.Label(window, text=test_result)
test_result_label.pack()

# Create a button to run the clustering algorithm
run_button = tk.Button(window, text="Run Clustering", command=run_clustering)
run_button.pack()

# Run the Tkinter event loop
window.mainloop()

```

Output:

K-Means Clustering

Number of Clusters:
3

Cluster Center Points (comma-separated):
3.45,2.98,4.0

Test Number:
3.7

Run Clustering

Final Clusters:
Cluster 1: [3.45, 3.24]
Cluster 2: [2.98]
Cluster 3: [3.78, 4.0, 3.9]

Test point 3.7 belongs to cluster [3]

Fig 8.1: User interface of the solution k-means algorithm with 3 clusters.

K-Means Clustering

Number of Clusters:
3

Cluster Center Points (comma-separated):
3.45,2.98,4.0

Test Number:
3.2

Run Clustering

Final Clusters:
Cluster 1: [3.45, 3.24]
Cluster 2: [2.98]
Cluster 3: [3.78, 4.0, 3.9]

Test point 3.2 belongs to cluster [1]

Fig 8.2: User interface of the solution k-means algorithm with 3 clusters.

9. Write a program to implement Edit Distance algorithm. Test the program with ‘exercised’ and ‘executed’.

Ans: The Edit Distance algorithm determines the minimum number of changes (insertions, deletions, or replacements) required to change one string into another. It is a dynamic programming algorithm. In order to make the first n characters of the first string into the first m characters of the second string, the Edit Distance algorithm builds a table that contains the least amount of changes necessary. One row at a time, beginning with the top row, is filled in on the table. The value in the bottom right cell of the table is the bare minimal amount of changes necessary to change the two strings.

Implementation of the program: A python program to implement the Edit Distance algorithm with test the program with ‘exercised’ and ‘executed’ is given below:

```

import tkinter as tk
def calculate_edit_distance():
    #exercised executed
    string1 = entry1.get()
    string2 = entry2.get()
    distance = edit_distance(string1, string2)
    result.config(text="Edit Distance: " + str(distance))
def edit_distance(string1, string2):
    m = len(string1)
    n = len(string2)
    dp = [[0 for x in range(n + 1)] for x in range(m + 1)]
    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            elif string1[i-1] == string2[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = 1 + min(dp[i][j-1], dp[i-1][j], dp[i-1][j-1])
    return dp[m][n]
root = tk.Tk()

```

```

root.title("Edit Distance Calculator")
label1 = tk.Label(root, text="Enter first string:")
entry1 = tk.Entry(root)
label2 = tk.Label(root, text="Enter second string:")
entry2 = tk.Entry(root)

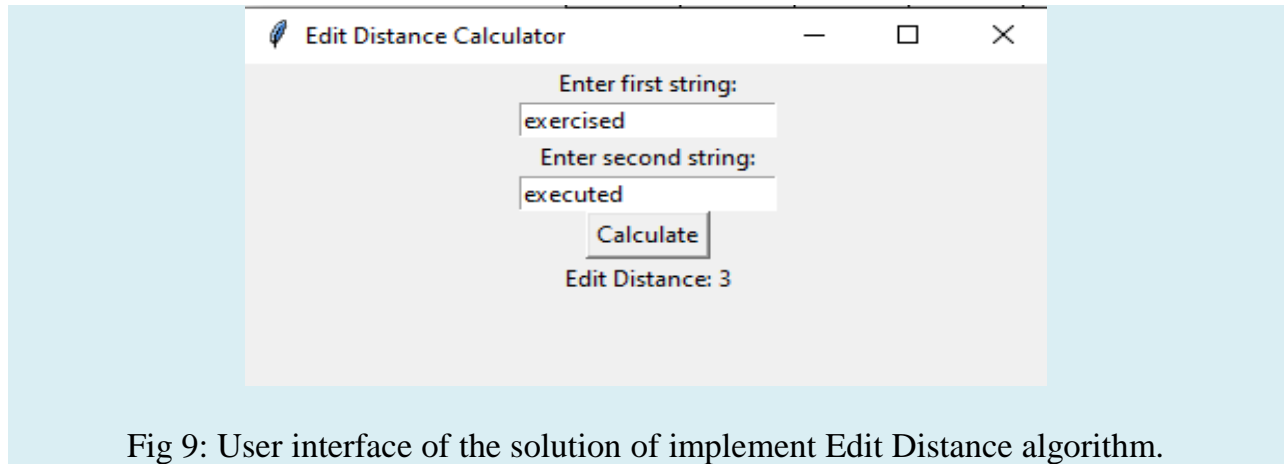
```

```

calculate_button = tk.Button(root, text="Calculate",
command=calculate_edit_distance)
result = tk.Label(root, text="")
label1.pack()
entry1.pack()
label2.pack(); entry2.pack()
calculate_button.pack()
result.pack()
root.mainloop()

```


Output:



10. Write a program to solve to predict the T-shirt size of Mr. Perfect, whose height is 161 cm and his weight is 61 kg using KNN algorithm with $k = 3$ or 5. You may use either Euclidean or Manhattan as a distance function.

Ans: A straightforward and uncomplicated supervised machine learning technique called K-nearest neighbors (KNN) may be utilized for both classification and regression applications. In order to predict the label of a new instance based on the labels of its k closest neighbors, the KNN algorithm first identifies the k examples that are most similar to a new instance.

Implementation of the program: A python program to implement the given problem is given below:

```
import numpy as np
import pandas as pd
import tkinter as tk
```

```

# Define the data
data = [(158, 58, 'M'), (158, 59, 'M'), (158, 63, 'M'), (160, 59, 'M'), (160, 60, 'M'),
(163, 60, 'M'), (163, 61, 'M'), (160, 64, 'L'), (163, 64, 'L'), (165, 61, 'L'), (165, 62, 'L'),
(165, 65, 'L'), (168, 62, 'L'), (168, 63, 'L'), (168, 66, 'L'), (170, 63, 'L'), (170, 64, 'L'),
(170, 68, 'L')]

# Convert data to a dataframe
df = pd.DataFrame(data, columns=['height', 'weight', 'tshirt_size'])

# Define the tkinter window
window = tk.Tk()
window.title("T-Shirt Size Prediction")

# Create input labels and entry widgets
height_label = tk.Label(window, text="Enter height in cms:")
height_label.pack()
height_entry = tk.Entry(window)
height_entry.pack()
weight_label = tk.Label(window, text="Enter weight in kgs:")
weight_label.pack()
weight_entry = tk.Entry(window)
weight_entry.pack()
k_label = tk.Label(window, text="Enter the value of k:")

```

```

k_label.pack()
k_entry = tk.Entry(window)
k_entry.pack()
# Define the button click function
def predict_tshirt_size():
    height = int(height_entry.get())
    weight = int(weight_entry.get())
    k = int(k_entry.get())
# Define the target observation
target = np.array([height, weight])
# Calculate the Euclidean distance
def euclidean_distance(x, y):
    return np.sqrt(np.sum((x - y) ** 2))
# Define the KNN algorithm
def knn(df, target, k):
    distances = []
    for i, row in df.iterrows():
        distance = euclidean_distance(np.array([row['height'], row['weight']]),
target)
        distances.append((row['tshirt_size'], distance))

```

```

distances = sorted(distances, key=lambda x: x[1])
return [x[0] for x in distances[:k]]
result = knn(df, target, k)
predicted_size = max(set(result), key=result.count)
result_label.config(text="Predicted T-shirt size: " + predicted_size)
predict_button = tk.Button(window, text="Predict T-Shirt Size",
command=predict_tshirt_size)
predict_button.pack()
# Create a label to display the prediction result
result_label = tk.Label(window, text="")
result_label.pack()
# Start the tkinter event loop
window.mainloop()

```

Output:

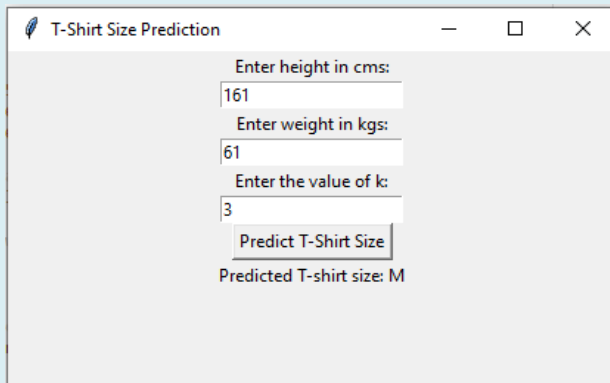


Fig 10.1: User interface of the solution of KNN at k = 3.

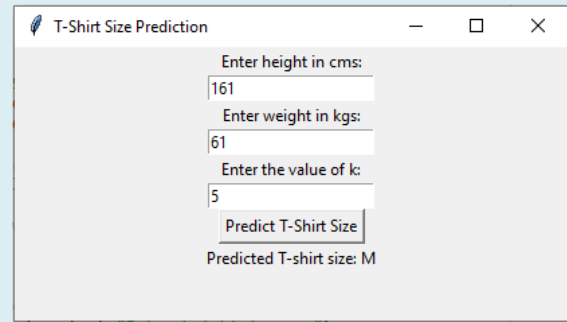


Fig 10.2: User interface of the solution of KNN at k = 5.

11. Potato vs tomato image classification problem:

Ans: In order to recognize the objects or scenes in an image, a model must be trained for the machine learning job known as "image classification." Face recognition, object identification, and scene comprehension are just a few applications for this. Various techniques, including CNN, SVM, random forest, and others, can be utilized for picture classification tasks. Here, we categorize a collection of photographs of potatoes and tomatoes using the support vector machine (SVM) technique. Support vector machines (SVMs) are a form of supervised learning technique that may be applied to both regression and classification applications. Finding a hyperplane that divides two classes

of data points is the foundation of SVMs. In order to maximize the margin between the two classes, the hyperplane is selected such that it is as far away from the nearest data points on either side as is feasible. This contributes to the model's robustness to data noise and outliers. Below are the steps we take to do picture categorization using SVM:

1. Gather an image dataset: Using a smartphone camera and web scraping on Google, we gather photographs of potatoes and tomatoes.
2. Extraction of features from the photos: We make use of color characteristics, such as the pixel values on the three channels of the images. Here, the picture name includes the labeling.
3. Create and test an SVM model: Using the train-test split approach, we created an SVM model and trained it using the dataset we had gathered. The test data displays a 100% accuracy result.
4. Utilize the model to categorize fresh images: We utilize user-inputted image to estimate its class label using the model.

Implementation of the program: Here a python program to classify tomatoes and potatoes images is given below:

```
import os
import tkinter as tk
from tkinter import filedialog, messagebox
from PIL import Image
import numpy as np
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
import joblib

# Path to folder containing images to train the SVM
image_folder = 'D:/8th semister/machine learning/tomato-potato-dataset/image'
# Function to extract features from the image
def extract_features(image_path):
    image = Image.open(image_path)
    image = image.resize((224, 224)) # Resize the image to a fixed size
    image_array = np.array(image)
    flattened_image = image_array.flatten() # Flatten the image into a 1D array
    return flattened_image
```

```

def load_data(image_folder):
    images = []
    labels = []
    class_names = []
    class_mapping = {} # Mapping between class labels and indices
    # Traverse the image folder
    for root, dirs, files in os.walk(image_folder):
        for file in files:
            if file.lower().endswith(('.jpg', '.jpeg', '.png')):
                image_path = os.path.join(root, file)
                label = file.split("_")[0]
                if label not in class_mapping:
                    class_mapping[label] = len(class_mapping)
                    class_names.append(label)
                images.append(extract_features(image_path))
                labels.append(class_mapping[label])
    return np.array(images), np.array(labels), class_names

def train_svm(images, labels):
    scaler = StandardScaler()
    scaled_images = scaler.fit_transform(images) # Scale the image features
    X_train, X_test, y_train, y_test = train_test_split(scaled_images, labels,
test_size=0.2, random_state=42)
    classifier = svm.SVC(kernel='linear')

    classifier.fit(X_train, y_train)
    y_pred = classifier.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print("Accuracy:", accuracy)
    return classifier, scaler

def classify_image(image_path, classifier, scaler, class_names):
    image_features = extract_features(image_path)
    scaled_features = scaler.transform([image_features]) # Scale the image
features
    predicted_class_index = classifier.predict(scaled_features)[0]
    predicted_label = class_names[predicted_class_index]
    return predicted_label

images, labels, class_names = load_data(image_folder)
classifier, scaler = train_svm(images, labels)

def classify_button_click():
    file_path = filedialog.askopenfilename(initialdir="/", title="Select Image File")
    if file_path:

```

```

predicted_label = classify_image(file_path, classifier, scaler, class_names)
result_label.config(text=f"Predicted Class: {predicted_label}")
else:
    messagebox.showwarning("Image Classification", "No file selected.")
# Create the GUI window
window = tk.Tk()
window.title("Image Classification")
window.geometry("300x150")
# Create a classify button
classify_button = tk.Button(window, text="Classify Image",
command=classify_button_click)
classify_button.pack()

# Create a label to display the result
result_label = tk.Label(window, text=" ")
result_label.pack()
# Run the GUI event loop
window.mainloop()

```

Output:

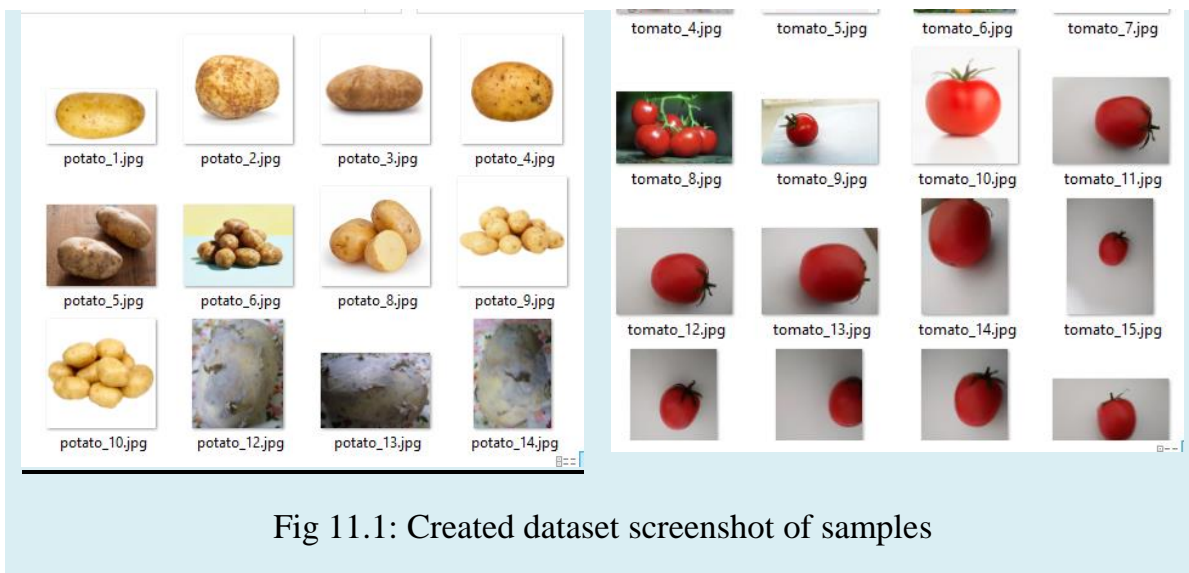


Fig 11.1: Created dataset screenshot of samples

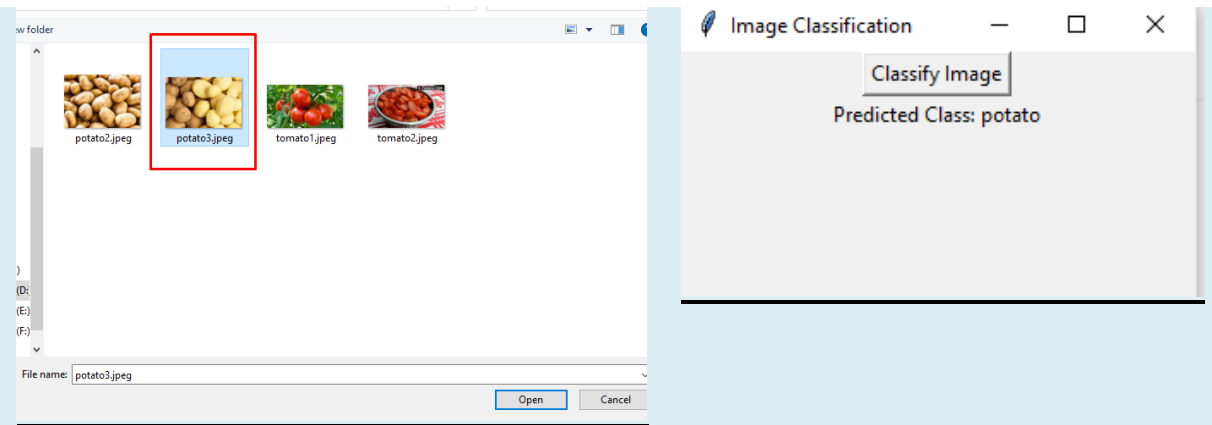


Fig 11.1: Test image with result of potato from user using User Interface

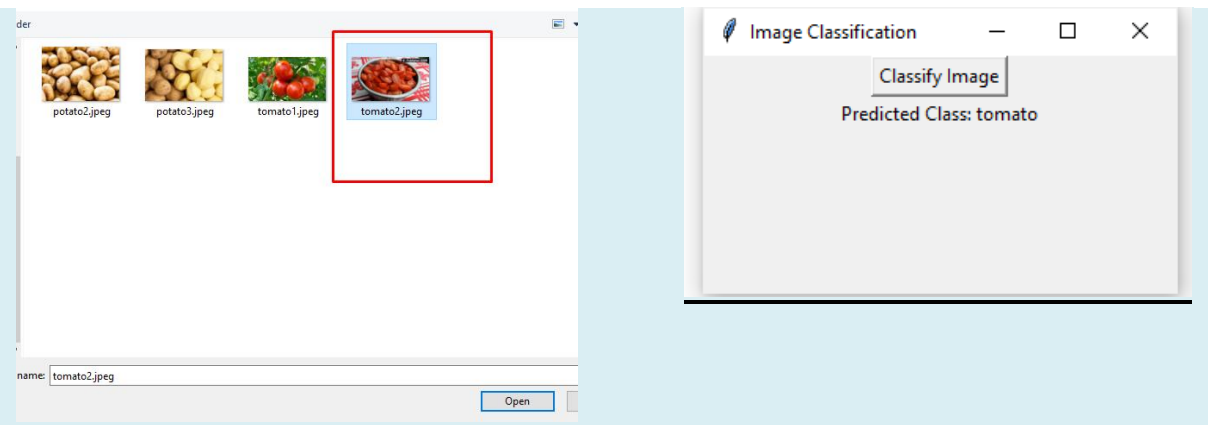


Fig 11.2: Test image with result of tomato from user using User Interface

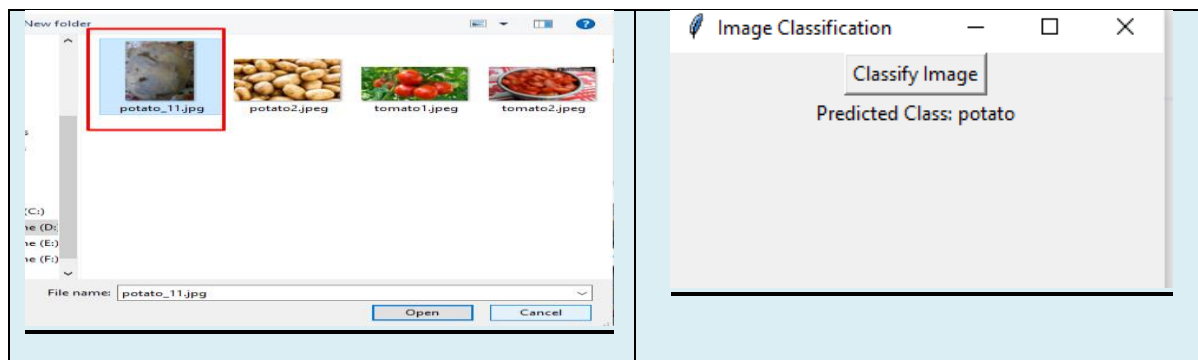


Fig 11.3: Test image with result of potato from user using User Interface