SRI VENKATESWARA COLLEGE OF ENGINEERING & TECHNOLOGY

Autonomous

R.V.S. Nagar, Chittoor-517127.

LAB MANUAL

SUB NAME: COMPUTER VISION & MACHINE LEARNING LAB **(23ACA14)**

PREPARED BY:

ANJANEYULU.K, M.Tech (CSE), MBA (Finance).,
Assistant Professor, Dept. of CSM

Sujitha, M.Tech(CSE)
Assistant Professor, Dept. of CSM

Sri Venkateswara College of Engineering & Technology

Course Objectives:

- To impart practical knowledge of computer vision concepts using OpenCV and image processing libraries.
- To implement core machine learning algorithms and evaluate model performance.
- To work with real-world datasets for classification, regression, and image processing tasks.
- To train, test, and validate models using Python, TensorFlow, and Scikit-learn.
- To understand the integration of ML models in vision applications.

Course Outcomes:

After successful completion of this lab, students will be able to:

- Apply computer vision techniques to solve real-time image processing problems. **(Apply - L3)**
- Train and evaluate machine learning models for classification and regression tasks. **(Analyze - L4)**
- Design and test feature extraction techniques from images. **(Create - L6)**
- Use OpenCV, Scikit-learn, TensorFlow/PyTorch for practical implementations. **(Apply - L3)**
- Integrate vision-based features with ML algorithms for end-to-end solutions. **(Evaluate - L5)**

List of Experiments

1.Image preprocessing techniques: resizing, filtering, thresholding using OpenCV

2. Edge detection using Sobel, Canny, and Laplacian operators

3. Object detection using contour detection and bounding boxes

4. Feature extraction using HOG, SIFT, and ORB

5. Implement face detection using Haar cascades or DNN models

6. Train a machine learning model (SVM / KNN) for image classification

7. Build and evaluate a decision tree classifier using scikit-learn

8. Implement a logistic regression model for binary classification on numerical dataset

9. Apply PCA for feature reduction and visualization

10. Design a simple neural network using TensorFlow/Keras for image classification

11. Train and evaluate a CNN model for digit recognition using MNIST dataset

12. Real-time emotion recognition using webcam input and pre-trained model integration

Experiment :1

Problem statement:

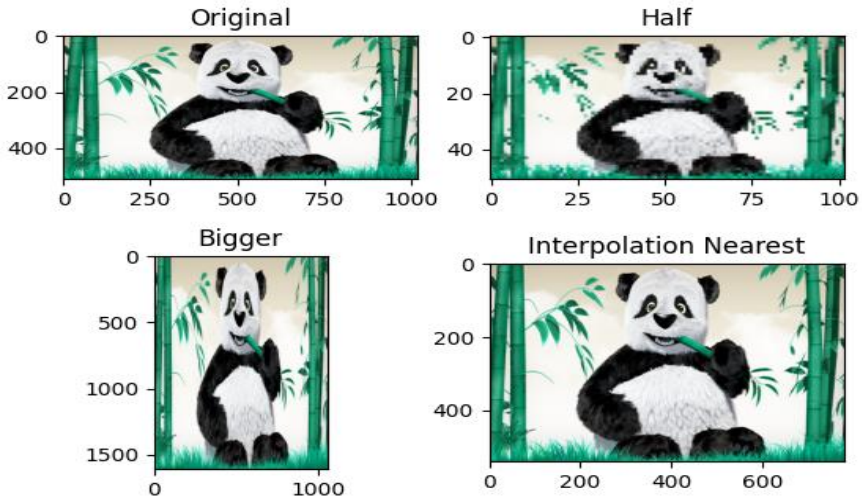
1. **Write a python program for Image preprocessing techniques using OpenCV:**
   a) **resizing**
   b) **filtering**
   c) **thresholding**

1a) Aim: to resize the image

PROGRAM:

```
import cv2
import matplotlib.pyplot as plt

image = cv2.imread('D:/DOLL.jpg',1)
# Loading the image

half = cv2.resize(image, (0, 0), fx = 0.1, fy = 0.1)
bigger = cv2.resize(image, (1050, 1610))

stretch_near = cv2.resize(image, (780, 540),
               interpolation = cv2.INTER_LINEAR)


Titles =["Original", "Half", "Bigger", "Interpolation Nearest"]
images =[image, half, bigger, stretch_near]
count = 4

for i in range(count):
    plt.subplot(2, 2, i + 1)
    plt.title(Titles[i])
    plt.imshow(images[i])

plt.show()
```

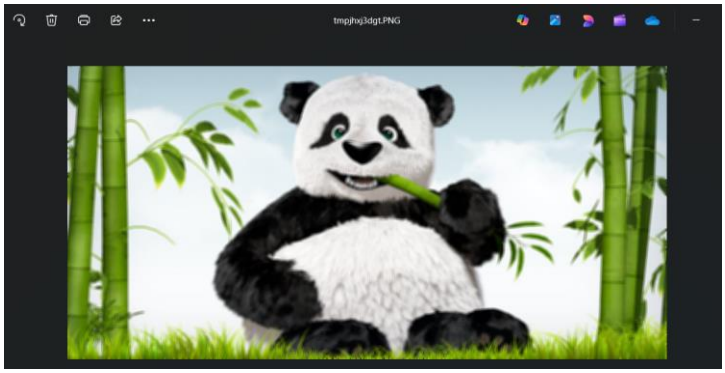
OUTPUT:

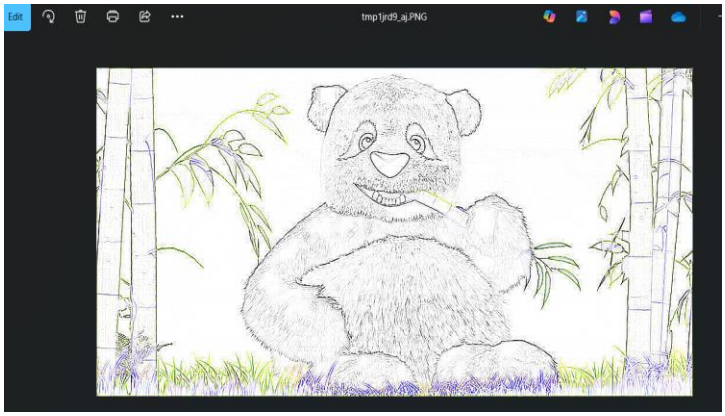


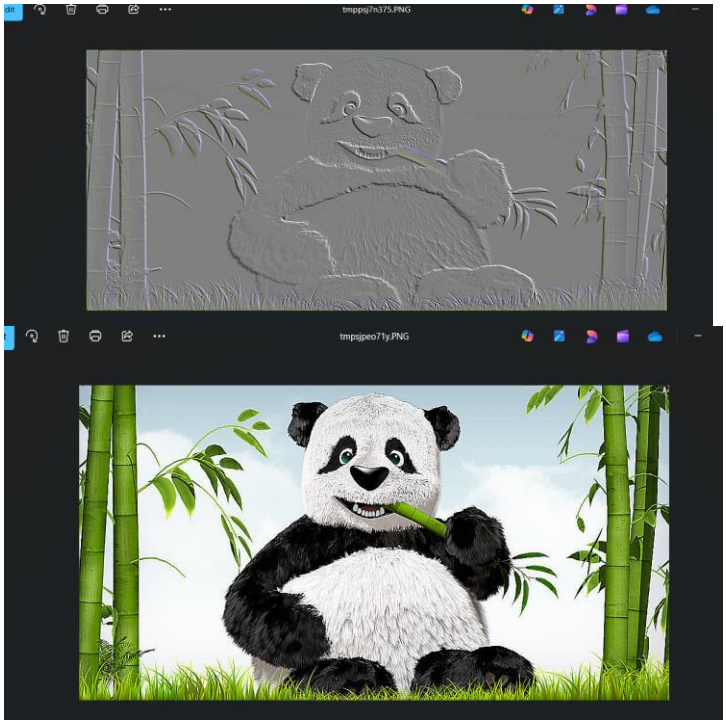
1 b) Aim: to perform image filter operations.
PROGRAM:

```
# Importing Image and ImageFilter module from PIL package
from PIL import Image, ImageFilter

# creating a image object
im1 = Image.open('D:/DOLL.jpg')

# applying the blur filter
im2 = im1.filter(ImageFilter.BLUR)
im3 = im1.filter(ImageFilter.CONTOUR)
im4 = im1.filter(ImageFilter.EMBOSS)
im5 = im1.filter(ImageFilter.EDGE_ENHANCE)
im6 = im1.filter(ImageFilter.EDGE_ENHANCE_MORE)

im2.show()
im3.show()
im4.show()
im5.show()
im6.show()
```

output:







1c) Aim: to perform image thresholding operations

PROGRAM:

```
import cv2

# Loading the image named test.jpg
img = cv2.imread('D:/DOLL.jpg')

# Converting color mode to Grayscale
# as thresholding requires a single channeled image
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
ret, thresh = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY)
ret, thresh1 = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY_INV)
ret, thresh2 = cv2.threshold(img, 127, 255, cv2.THRESH_TRUNC)
ret, thresh3 = cv2.threshold(img, 127, 255, cv2.THRESH_TOZERO)
ret, thresh4 = cv2.threshold(img, 127, 255, cv2.THRESH_TOZERO_INV)
# Displaying the output image
cv2.imshow('Image', img)
cv2.imshow('Binary Threshold', thresh)
# Displaying the output image
cv2.imshow('Binary Inverse Threshold', thresh1)
# Displaying the output image
cv2.imshow('Truncate Threshold', thresh2)
# Displaying the output image
cv2.imshow('Truncate Threshold', thresh3)
# Displaying the output image
cv2.imshow('Truncate Threshold', thresh4)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Result: this program has been executed successfully and image processing operations are performed using opencv package.

EXPERIMENT 2

AIM: Write a python program to implement edge detection using Sobel, Canny, and Laplacian operators

Algorithm

Sobel Edge Detection Algorithm

Step1: Read the input image.
Step 2: If the image is colored convert it into gray scale image
Step3: Apply Gaussian Blur to reduce noise
Step 4: Calculate Gx and Gy to detects edges for both horizontal and vertical direction.
Step 5: Compute Gradient Magnitude by using $G=\sqrt{(Gx^2+Gy^2)}$
Step 6: Apply a threshold to highlight strong edges
Step 7: Display or save the edge-detected image

Canny Edge Detection Algorithm

Step1: Read the input image.
Step 2: If the image is colored convert it into gray scale image
Step3: Apply Gaussian Blur to reduce noise
Step 4: Use **Sobel operator** to compute intensity gradients and calculate Gx and Gy to detects edges for both horizontal and vertical direction.
Step 5: Compute Gradient Magnitude by using $G=\sqrt{(Gx^2+Gy^2)}$
Step 6: Thin out edges by keeping only local maxima in the gradient direction. Apply a threshold to highlight strong edges.
Step 7: To classify edges define **low** and **high thresholds**.
(a) Strong edges (above high threshold)
(b) Weak edges (between low and high threshold)
(c) Suppressed (below low threshold).

Step 8: Keep strong edges, Connect weak edges if they are connected to strong edges and discard the isolated weak edges.
Step 9: Display or save the edge-detected image

Laplacian Edge Detection Algorithm

Step1: Read the input image.
Step 2: If the image is colored convert it into gray scale image
Step3: Apply Gaussian Blur to reduce noise
Step 4: Apply Laplacian operator by using discrete laplacian kernel.

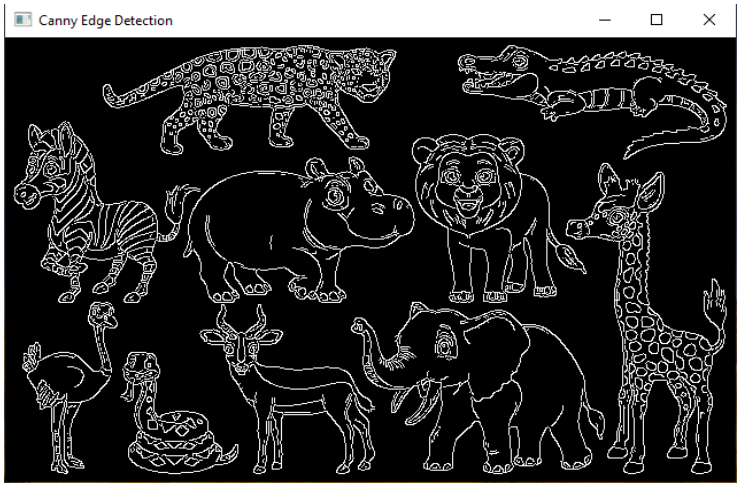
$$\begin{vmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{vmatrix} \qquad \begin{vmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{vmatrix}$$

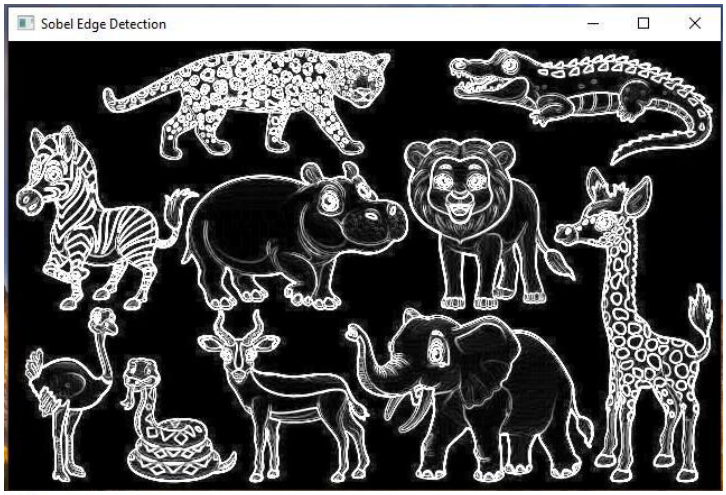
Step 5: Take the absolute value of results to avoid negative intensities
Step 6: Apply a threshold to highlight strong edges.
Step 7: Display or save the edge-detected image.

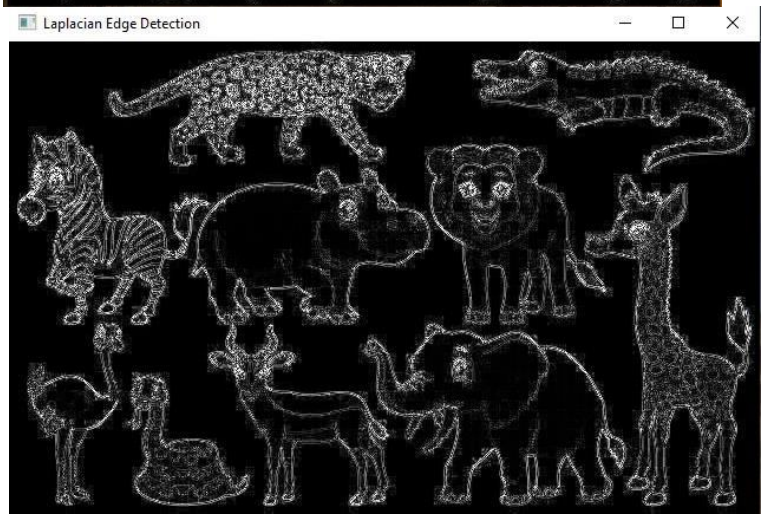
Program

```
import cv2
import numpy as np
# Load the image
image = cv2.imread("C:\\cvip\\animal.jpg") # Replace with your image path
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# Sobel Edge Detection
sobel_x = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3) # X direction
sobel_y = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3) # Y direction
sobel_combined = cv2.magnitude(sobel_x, sobel_y) # Combine both
# Canny Edge Detection
canny = cv2.Canny(gray, 100, 200)
# Laplacian Edge Detection
laplacian = cv2.Laplacian(gray, cv2.CV_64F)
# Convert results to uint8
sobel_combined = cv2.convertScaleAbs(sobel_combined)
laplacian = cv2.convertScaleAbs(laplacian)
# Display All
cv2.imshow('Original Image', image)
cv2.imshow('Sobel Edge Detection', sobel_combined)
cv2.imshow('Canny Edge Detection', canny)
cv2.imshow('Laplacian Edge Detection', laplacian)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output


Canny Edge Detection


Sobel Edge Detection


Laplacian Edge Detection

EXPERIMENT 3

Problem Staement : write a python program to implement Object detection using contour detection and bounding boxes on an image.
Aim : to implement Object detection using contour detection and bounding boxes on an image.

Algorithm :
Step 1: Load Image
1. Read the image from file using a library (e.g., OpenCV).

Step 2: Preprocessing
3. Convert the image to grayscale to reduce computational complexity.
4. Apply Gaussian blur to the grayscale image to smooth it and remove noise.
5. Perform edge detection using the Canny algorithm to highlight the object boundaries.

Step 3: Contour Detection
6. Use cv2.findContours() to find contours from the edge-detected image.
7. Retrieve only external contours to avoid nested detections.

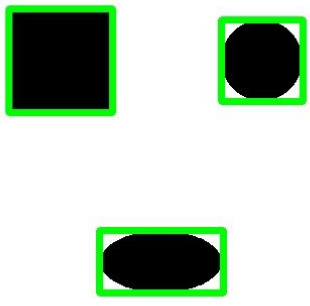
Step 4: Filter and Draw Bounding Boxes
8. For each detected contour:
   Check if the contour area is above a minimum threshold to eliminate noise (e.g., area > 50).
   Use cv2.boundingRect() to compute the smallest rectangle that bounds the contour.
   Draw a rectangle on the original image using cv2.rectangle().

Step 5: Display or Save Output
9. Show or save the resulting image with bounding boxes drawn around detected objects.

Program:
```
import cv2
import numpy as np

# Load image
image = cv2.imread(r"C:\cvip\ts.jpg")
if image is None:
    print("Image not found.")
    exit()

# Make a copy of the original for display
original_image = image.copy()

# Preprocessing
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(gray, (5, 5), 0)
edges = cv2.Canny(blurred, 30, 100)

# Find contours
contours, _ = cv2.findContours(edges.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
print(f"Number of contours detected: {len(contours)}")
```

```
# Draw bounding boxes on the image
for contour in contours:
    if cv2.contourArea(contour) < 50:
        continue
    x, y, w, h = cv2.boundingRect(contour)
    cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)

# Concatenate original and result images side-by-side
side_by_side = np.hstack((original_image, image))

# Add a header row with labels
header_height = 40
header = np.ones((header_height, side_by_side.shape[1], 3), dtype=np.uint8) * 255

# Write titles on the header
font = cv2.FONT_HERSHEY_SIMPLEX
font_scale = 0.8
color = (0, 0, 0)
thickness = 2

# Get width of each image
half_width = original_image.shape[1]

cv2.putText(header, "Original Image", (int(half_width * 0.25) - 60, 30), font, font_scale,
color, thickness)
cv2.putText(header, "Detected Image", (int(half_width * 1.25) - 60, 30), font, font_scale,
color, thickness)

# Combine header and side-by-side images
final_output = np.vstack((header, side_by_side))

# Show the final image
cv2.imshow("Original and Detected", final_output)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output
Original image



EXPERIMENT 4

Problem Statement: Write a python program to implement Feature extraction using HOG, SIFT, and ORB.

4 (a). Aim : to implement Feature extraction using HOG, SIFT, and ORB

Algorithm:

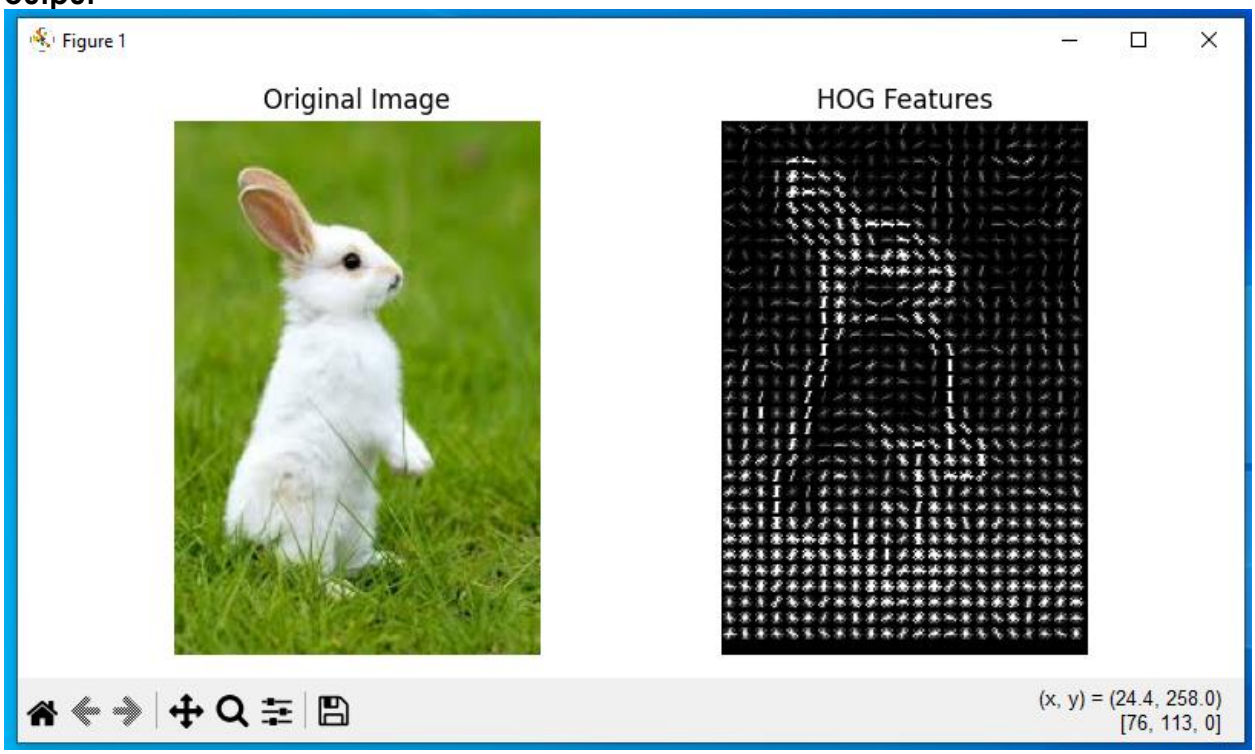
Step 1: Load image and convert to grayscale (or use single channel).
Step 2: Optionally apply gamma / contrast normalization (improves robustness).
Step 3: Compute image gradients Gx and Gy (e.g., Sobel filter).
Step 4: Compute gradient magnitude m = sqrt(Gx^2 + Gy^2) and orientation θ = atan2(Gy, Gx) (usually map orientation to 0–180° for unsigned HOG).
Step 5: Divide image into small cells (e.g., 8×8 pixels).
Step 6: For each cell, build an orientation histogram (e.g., 9 bins) by voting each pixel's magnitude into neighboring orientation bins (and often spatially bilinear weight into adjacent cells to smooth).
Step 7: Group adjacent cells into overlapping blocks (e.g., 2×2 cells per block). For each block, concatenate its cells' histograms and apply block normalization (L2, L2-Hys, or L1-sqrt).
Step 8: Slide the block window across the image (blocks overlap) and collect the normalized histograms.
Step 9: Concatenate all block vectors into a single HOG feature vector for the window/image patch.
Step 10: (For detection) feed the HOG vector to a classifier (commonly linear SVM). For full image detection use a sliding window & image pyramid.
Step 11: End the program

Program

```
from skimage.io import imread
from skimage.transform import resize
from skimage.feature import hog
from skimage import exposure
import matplotlib.pyplot as plt

# 1. Load the image
img = imread(r"C:\\cvip\\rabbit.jpg")  # Replace with your path
# 2. Resize the image
#resized_img = resize(img, (128, 64))
# 3. Compute HOG features
fd, hog_image = hog(img, orientations=9, pixels_per_cell=(8, 8),
                    cells_per_block=(2, 2), visualize=True, channel_axis=-1)
# 4. Enhance HOG image contrast
hog_image_rescaled = exposure.rescale_intensity(hog_image, in_range=(0, 10))

# 5. Display
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4), sharex=True, sharey=True)
ax1.imshow(img, cmap=plt.cm.gray)
ax1.set_title('Original Image')
ax1.axis('off')

ax2.imshow(hog_image_rescaled, cmap=plt.cm.gray)
ax2.set_title('HOG Features')
```

```
ax2.axis('off')
plt.tight_layout()
plt.show()
print(f"HOG feature descriptor shape: {fd.shape}")
```

output



4(b) Aim: to implement feature extraction using SIFT

Algorithm

Step 1: Start

Read input image .
If color, convert to grayscale G.
Convert to floating point (e.g., float32) and optionally normalize to [0,1].

Step 2 : Build Gaussian scale-space

Let k = 2^(1/num_intervals).
For each octave o = 0..num_octaves-1:
For each interval i = 0..num_intervals+2 (make num_intervals+3 Gaussian images per octave):

- Compute σ_{o,i} = σ * k^{i} * 2^{o} (or apply incremental Gaussian blur).
- Produce blurred image L_{o,i} = GaussianBlur(image_at_octave_o, σ_{o,i}).

(Optionally) downsample image by factor 2 to start next octave.

Step 3: Compute Difference-of-Gaussians (DoG)

For each octave o, compute D_{o,i} = L_{o,i+1} - L_{o,i} for i = 0..num_intervals+1..

Step 4: Detect scale-space extrema (candidate keypoints)

For each pixel (x,y) in D_{o,i}:

- Compare pixel value to its 26 neighbors (3×3 in current scale, plus 3×3 in scale above and below).

- If it is a strict local maximum or minimum, mark as a candidate keypoint.

Step 5: Accurate keypoint localization (subpixel & filtering)

For each candidate:

- Fit a 3D quadratic (Taylor expansion) to the local sample points to estimate subpixel/subscale location (solve for offset).
- If the offset is large, move to the interpolated location and repeat; if it cannot be localized reliably, discard.

Step 6: Assign orientations to keypoints

- Compute gradient magnitude m(x,y) and orientation θ(x,y) for samples in a circular neighborhood centered at (x_k,y_k). The neighborhood radius is
- Build an orientation histogram with 36 bins (covering 360°).

Step 7 : Compute descriptor for each oriented keypoint

For every keypoint with assigned orientation:

(a) Rotate the coordinate frame of the local neighborhood to align with the keypoint orientation.

(b) Consider a square region around the keypoint, typically sized 16 × 16 samples (scaled by keypoint scale).

(c) Divide this region into 4 × 4 subregions (cells). In each cell compute an 8-bin orientation histogram of the gradient samples (weighted by magnitude and a Gaussian centered on the keypoint).

(d) Concatenate the 16 histograms → a 4×4×8 = 128-dimensional vector.

Step 8: Collect final keypoints & descriptors

(1) For each retained keypoint (location, scale, orientation) produce its 128-D descriptor.

(2) Output the set: {(x,y,scale,orientation), descriptor}.

Step 9: Post-processing (optional)

Optionally apply non-maximum suppression across neighboring keypoints to remove redundancies.

Optionally filter by keypoint strength or enforce a maximum number of keypoints.

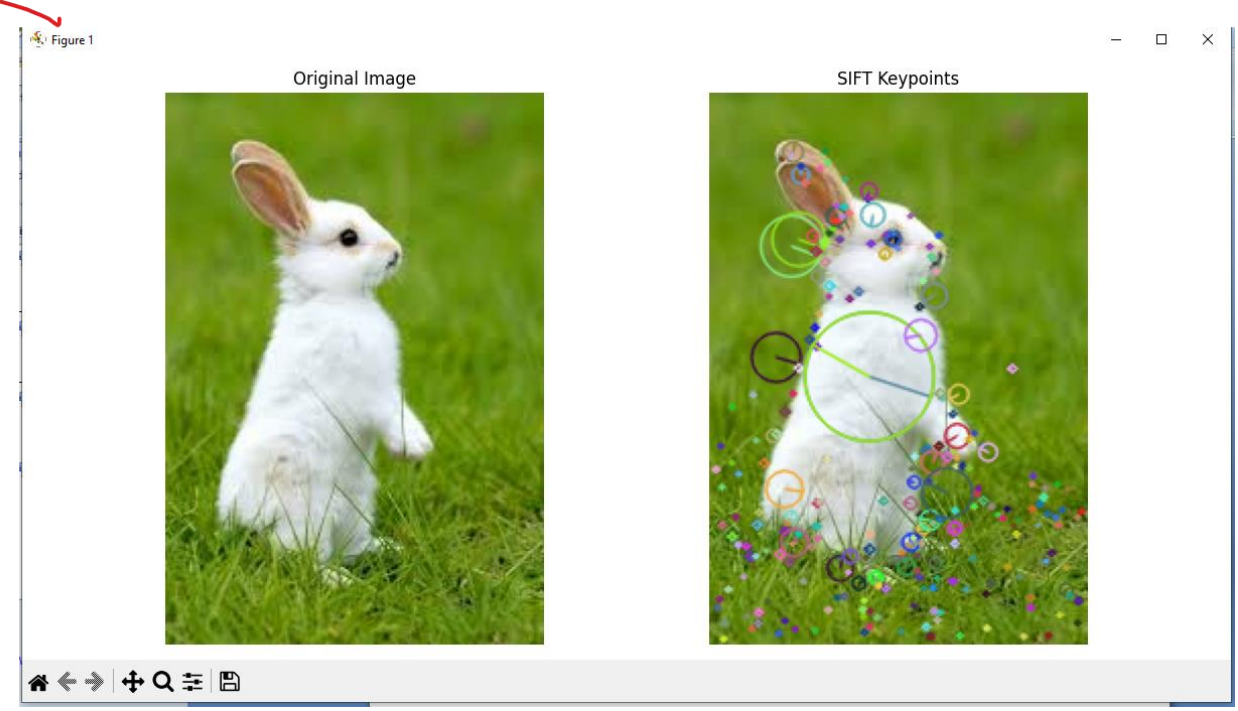
Step 10: Display result

program

```
from skimage.io import imread
from skimage.transform import resize
from skimage.feature import hog
from skimage import exposure
import matplotlib.pyplot as plt
import cv2

# 1. Load the image
image = cv2.imread(r"C:\cvip\rabbit.jpg")  # Replace with your image path
if image is None:
    print("Image not found.")
    exit()
# Convert to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# 2. Create SIFT detector
sift = cv2.SIFT_create()

# 3. Detect keypoints and compute descriptors
keypoints, descriptors = sift.detectAndCompute(gray, None)
```

```
print("Number of keypoints detected:", len(keypoints))
print("Descriptor shape:", descriptors.shape)  # (number of keypoints, 128)

# 4. Draw keypoints on the image
sift_image = cv2.drawKeypoints(image, keypoints, None,
                               flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

# Convert BGR to RGB for Matplotlib
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
sift_image_rgb = cv2.cvtColor(sift_image, cv2.COLOR_BGR2RGB)
# 5. Display both images side-by-side
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))
ax1.imshow(image_rgb)
ax1.set_title("Original Image")
ax1.axis("off")
ax2.imshow(sift_image_rgb)
ax2.set_title("SIFT Keypoints")
ax2.axis("off")
plt.tight_layout()
plt.show()
```

output



4(c) Aim: to implement Feature extraction using ORB

Algorithm

Step 1: Start

Read input image, If color, convert to grayscale G (ORB works on single-channel) and convert to an appropriate numeric type (e.g., uint8).

Step 2: Build image pyramid

(a) Create an image pyramid with n_levels.
(b) Level 0 is the original image; each next level is downsampled by scaleFactor relative to the previous level.
(c) Purpose: provide scale invariance by detecting features at multiple scales.

Step 3: Detect FAST keypoint candidates at each pyramid level

- For each pyramid level l = 0..n_levels-1:
  a) Detect corner-like candidates using the FAST corner detector (with fastThreshold) on the scaled image.
  b) Record candidate coordinates in the coordinate system of that level.

Step 4: Keep strongest keypoints per level (score & non-maximum suppression)

(a) For each candidate compute a score:
  - Either FAST score or Harris response depending on scoreType.
(b) Apply non-maximum suppression in the local neighborhood to remove close weaker candidates.
(c) Optionally limit to a fair share per level (so keypoints distribute across scales).

Step 5: Adjust keypoint coordinates to original image scale

Map keypoint coordinates from their pyramid level back to the original image coordinates (multiply by appropriate scale).

Step 6: Retain top N keypoints

(a) Sort all retained keypoints (from all levels) by score.
(b) Keep top n_features keypoints overall.

Step 7: Compute orientation for each keypoint (intensity centroid)

Consider a circular patch around the keypoint (radius ~ patchSize/2 scaled by level).

Step 8: Compute BRIEF descriptors rotated by keypoint orientation (steered BRIEF)

(a) Use a predefined sampling pattern of point pairs inside the patchSize region (the BRIEF pattern).
(b) Rotate the sampling pattern by the keypoint orientation θ (this is the "steered" or rotated BRIEF).

Step 9: Store keypoints and descriptors

(a) For each keypoint store: (x, y, octave/level, scale, orientation, response/score).
(b) Store corresponding binary descriptor (e.g., uint8 array of length 32).

Step 10: Output

(a) Return the set {(keypoint_i), descriptor_i} for all retained keypoints.
(b) Typical descriptor matrix shape: (n_keypoints, 32).

Step 11: Optional post-processing / tips

(a) Filter or limit keypoints spatially if needed (grid-based allocation yields better coverage).
(b) For matching, use Hamming distance (binary descriptors). Example: BFMatcher(cv2.NORM_HAMMING).
(c) Cross-check, ratio test (adapted to Hamming), or RANSAC can filter false matches.

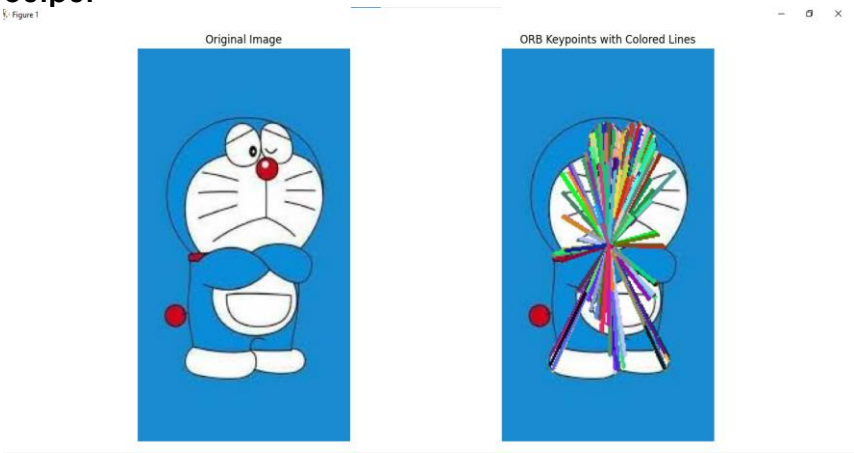
Step 12: Display the image

Program

```
from skimage.io import imread
from skimage.transform import resize
from skimage.feature import hog
from skimage import exposure
import matplotlib.pyplot as plt
import cv2
import numpy as np
# 1. Load the image
image = cv2.imread(r"C:\cvip\dora.jpg")  # Replace with your image path
if image is None:
    print("Image not found.")
    exit()

# Convert to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# 2. Create ORB detector
orb = cv2.ORB_create()

# 3. Detect keypoints and compute descriptors
keypoints, descriptors = orb.detectAndCompute(gray, None)

print("Number of keypoints detected:", len(keypoints))
print("Descriptor shape:", descriptors.shape)  # (number of keypoints, 32)

# Copy the original image for drawing
line_image = image.copy()

# Get image center
center_x = line_image.shape[1] // 2
center_y = line_image.shape[0] // 2

# 4. Draw random-colored lines from center to each keypoint
for kp in keypoints:
    x, y = int(kp.pt[0]), int(kp.pt[1])
    color = tuple(np.random.randint(0, 256, 3).tolist())  # Random color
    cv2.line(line_image, (center_x, center_y), (x, y), color, 2)

# Convert BGR to RGB for Matplotlib
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
line_image_rgb = cv2.cvtColor(line_image, cv2.COLOR_BGR2RGB)

# 5. Display both original and line-drawn images
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.imshow(image_rgb)
ax1.set_title("Original Image")
ax1.axis("off")

ax2.imshow(line_image_rgb)
```

```
ax2.set_title("ORB Keypoints with Colored Lines")
ax2.axis("off")

plt.tight_layout()
plt.show()
```

output



EXPERIMENT 5

Problem Statement: write a python program to implement face detection using Haar cascades or DNN models

AIM: to implement face detection using Haar cascades or DNN models

ALGORITHM

Step 1 : Start

Load model files (weights + architecture) into a DNN runtime (e.g., OpenCV DNN: readNet*, TensorFlow/PyTorch runtime, ONNX runtime). Verify model input requirements (size, color order, scale, mean, channel order).

Step 2: Read the input image I (BGR or RGB as required).

Step 3: Preprocess

Resize the input image, convert to float 32, apply scaling and convert into gray scale if the image is colored and finally swap the dimensions

(In OpenCV) use cv2.dnn.blobFromImage(image, scalefactor, size, mean, swapRB, crop=False).

Step 4: Inference

(a) Feed the blob into the network and run a forward pass.

(b) Receive raw outputs — format depends on model:

- SSD-like: detection array [batch, numDet, 7] with each: [image_id, class_id, confidence, x_min, y_min, x_max, y_max] (relative coords).
- YOLO-like: grid outputs requiring decoding (anchors, objectness scores, class probs).
- RetinaFace/MTCNN: detections + landmark outputs.

Step 5: Decode model outputs

(a) Convert relative coordinates to pixel coordinates: x = rel_x * image_width, etc.

(b) For YOLO: decode boxes from grid + anchors, compute objectness * class probability → final confidence.
(c) For multi-class models: filter only the 'face' class if the network is multi-class.

Step 6: Confidence filtering
Discard detections with confidence < conf_thresh.

Step 7: Non-Maximum Suppression (NMS)
Run NMS on remaining boxes (use confidence scores) to remove overlapping duplicates. Keep boxes with IoU < nms_iou_thresh.

Step 9: Optional size / aspect filtering
Remove boxes smaller than min_size or with implausible aspect ratios.

Step 10: Optional landmark detection & alignment
If model provides facial landmarks, extract them. Use landmarks to perform face alignment / crop normalized face patches for downstream tasks (recognition, expression analysis).

Step 11: Return results
Output final list of boxes with confidences (and optionally landmarks and the cropped face images). For visualization, draw bounding boxes + confidences and optionally landmarks.

Step 12: Batching & optimization (for speed)
(a) Batch multiple frames or crops if runtime supports batching. Use GPU or optimized runtimes (OpenVINO, TensorRT, ONNX Runtime with GPU).
(b) Use smaller input sizes or lighter models (e.g., Tiny-YOLO, MobileNet-SSD) for real-time.

Step 13: Display the image

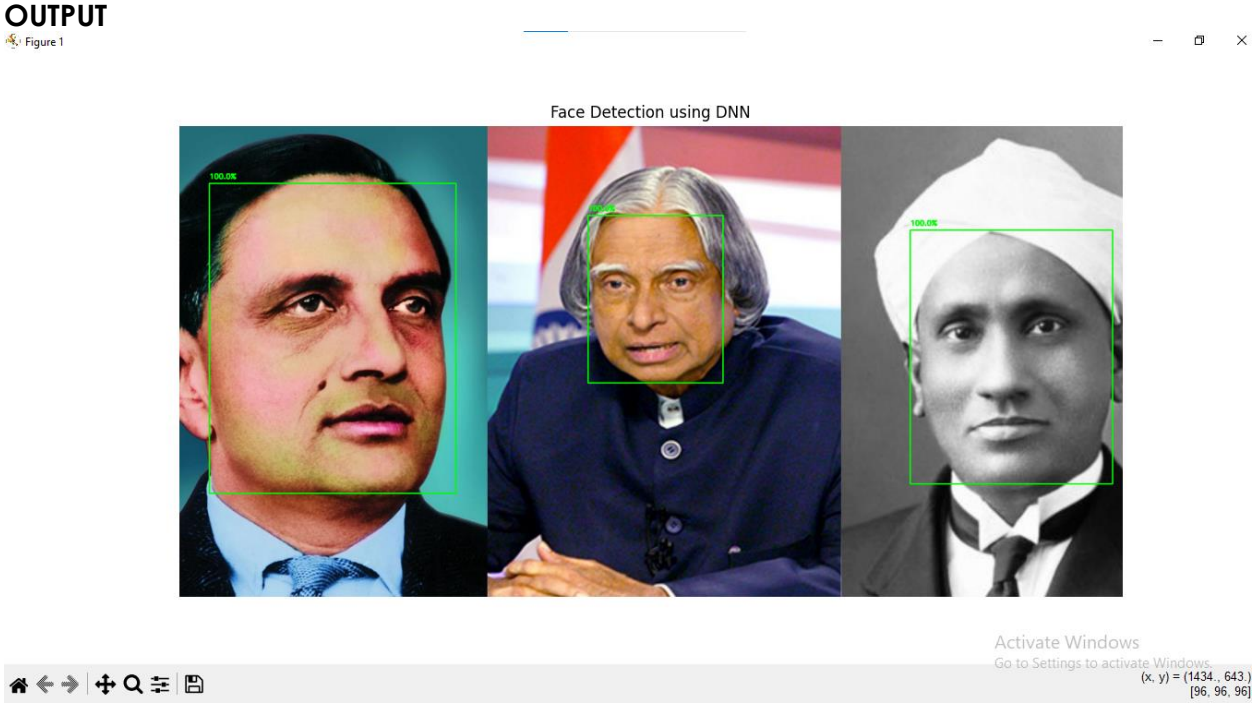
Paths to model files (download files from the given link)
https://github.com/sujigopidala/deploy.prototxt/blob/main/deploy.prototxt
https://github.com/sujigopidala/deploy.prototxt/blob/main/res10_300x300_ssd_iter_140000.caffemodel

PROGRAM

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# === Paths to model files (download these before running) ===
prototxt_path = r"C:\\cvip\\deploy.prototxt"
model_path = r"C:\\cvip\\res10_300x300_ssd_iter_140000_fp16.caffemodel"

# === Load the DNN model ===
net = cv2.dnn.readNetFromCaffe(prototxt_path, model_path)
# === Load image ===
image_path = r"C:\\cvip\\india.jpg"  # Change to your image path
image = cv2.imread(image_path)
if image is None:
    print("Image not found. Please check the path.")
    exit()
(h, w) = image.shape[:2]
# === Prepare blob for DNN ===
blob = cv2.dnn.blobFromImage(
    cv2.resize(image, (300, 300)),
```

```
    1.0,
    (300, 300),
    (104.0, 177.0, 123.0)
)
# === Perform detection ===
net.setInput(blob)
detections = net.forward()

# === Process detections ===
for i in range(0, detections.shape[2]):
    confidence = detections[0, 0, i, 2]
    # Filter weak detections
    if confidence > 0.5:
        box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
        (startX, startY, endX, endY) = box.astype("int")
        # Draw bounding box & label
        text = f"{confidence * 100:.1f}%"
        cv2.rectangle(image, (startX, startY), (endX, endY), (0, 255, 0), 2)
        y = startY - 10 if startY - 10 > 10 else startY + 10
        cv2.putText(image, text, (startX, y),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

# === Convert BGR to RGB for matplotlib display ===
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# === Show result ===
plt.imshow(image_rgb)
plt.axis("off")
plt.title("Face Detection using DNN")
plt.show()
```

OUTPUT



EXPERIMENT 6

Problem statement: Write a python program to train a machine learning model (SVM / KNN) for image classification.

AIM: to train a machine learning model (SVM / KNN) for image classification

Algorithm

Step 1: Organize data

(a) Arrange images in folders per class (e.g., data/cat/*.jpg, data/dog/*.jpg).
(b) Keep a separate test set (never touch during training/tuning).

Step 2: **Inspect & clean**

(a) Remove corrupted files, tiny images, and obvious mislabels.
(b) Note class counts (imbalance?) — plan for class weighting or resampling if needed.

Step 3: **Create splits**

(a) Split remaining data into train and validation (or use stratified K-fold CV).
(b) Typical: train 70–80%, val 10–15%, test 10–15. Use stratify=y.

Step 4: **Decide features**

(a) Apply the Handcrafted (HOG, color histograms), local descriptors (SIFT/ORB aggregated).
(b) CNN embeddings (recommended: pretrained ResNet/MobileNet penultimate layer).

Step 5: **Preprocess images**

(a) Resize to fixed size (e.g., 224×224 for CNN, 128×128 for HOG).
(b) Convert to required color space (grayscale for HOG; RGB for CNN).
(c) Apply any consistent normalization required by feature extractor.

Step 6: **Extract features**

Compute feature vectors for each image (one vector per image) and save features

Step 7: **Feature scaling**

Perform scaling for SVM StandardScaler (mean=0, std=1) **only** on training features; transform val/test with it.

Step 8: **Optional: dimensionality reduction**

If features are very high-dim, run PCA (fit on train only) to reduce dimensionality while preserving ~95% variance.

Step 9: **Build a pipeline**

Build scikit-learn Pipeline combining scaler, optional PCA, and SVC. This prevents leakage in cross-validation.

Step 10: **Hyperparameter tuning**

(a) Use GridSearchCV or RandomizedSearchCV with StratifiedKFold (k=5).
(b) Useful param grid:
    (1) C: [0.01, 0.1, 1, 10, 100]
    (2) kernel: ['linear','rbf']
    (3) gamma (for rbf): ['scale','auto', 1e-2, 1e-3]
    (4) class_weight: [None, 'balanced'] (if imbalance)
(c) Use scoring according to problem (accuracy, macro F1 for imbalanced multi-class).

Step 11: **Final evaluation**

Evaluate on held-out test set and calculate accuracy, confusion matrix, per-class precision/recall/F1. For binary, also ROC/AUC.

Step 12: **Error analysis**

(a) Inspect misclassified images per class.

(b) Look for dataset issues or confusing class pairs; consider augmentations or more data.

Step 13: **Save & deploy**

Save, deploy the model and continue the preprocessing → feature extraction → predict.

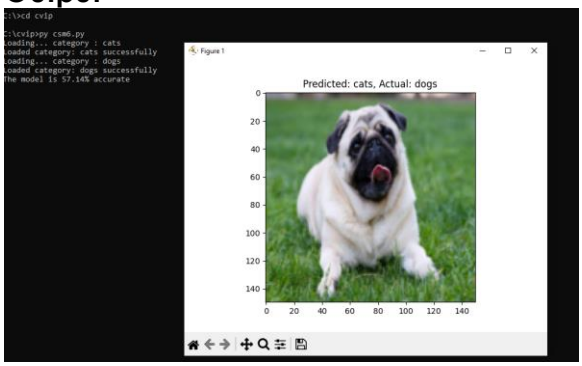
Step 14: Display the result

Program

```
import os
import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread
from skimage.transform import resize
from sklearn.model_selection import train_test_split
from sklearn import svm
from sklearn.metrics import accuracy_score

# Categories
Categories = ['cats', 'dogs']
flat_data_arr = []   # input array
target_arr = []      # output array
datadir = 'C:\\cvip\\test'  # path which contains the category folders

# Load images
for i in Categories:
    print(f'Loading... category : {i}')
    path = os.path.join(datadir, i)
    for img in os.listdir(path):
        img_array = imread(os.path.join(path, img))
        img_resized = resize(img_array, (150, 150, 3))  # resize images
        flat_data_arr.append(img_resized.flatten())
        target_arr.append(Categories.index(i))
    print(f'Loaded category: {i} successfully')

# Convert to numpy arrays
flat_data = np.array(flat_data_arr)
target = np.array(target_arr)

# Split into training and testing
x_train, x_test, y_train, y_test = train_test_split(
    flat_data, target, test_size=0.20, random_state=77, stratify=target
)

# Train model (SVM)
model = svm.SVC(kernel='linear')  # you can try 'rbf' too
model.fit(x_train, y_train)

# Predict on test set
y_pred = model.predict(x_test)

# Accuracy
```

```
accuracy = accuracy_score(y_test, y_pred)
print(f'The model is {accuracy*100:.2f}% accurate')

# (Optional) Test with one image
plt.imshow(x_test[0].reshape(150,150,3))
plt.title(f'Predicted: {Categories[y_pred[0]]}, Actual: {Categories[y_test[0]]}')
plt.show()
```

Output



Experiment :7 (use google colab)

Problem statement:
Write a python program to build and evaluate a decision tree classifier using scikit-learn.

Aim: To build and evaluate a decision tree classifier using scikit-learn.

Algorithm:

Step1: Start
Step2: Building the Decision Tree Classifier:

a) **Import necessary libraries:**
Import DecisionTreeClassifier from sklearn.tree, train_test_split from sklearn.model_selection, and metrics like accuracy_score, precision_score, recall_score, and f1_score from sklearn.metrics.
b) **Load or create dataset:** Prepare data with features (X) and target variable (y).
c) **Split the data:** Divide dataset into training and testing sets using train_test_split..
d) **Instantiate the classifier:** Create an instance of DecisionTreeClassifier. Specify hyperparameters like criterion (e.g., 'gini' or 'entropy'), max_depth, min_samples_split, etc.
e) **Train the model:** Fit the classifier to training data using the fit() method.

Step3: Evaluating the Decision Tree Classifier:

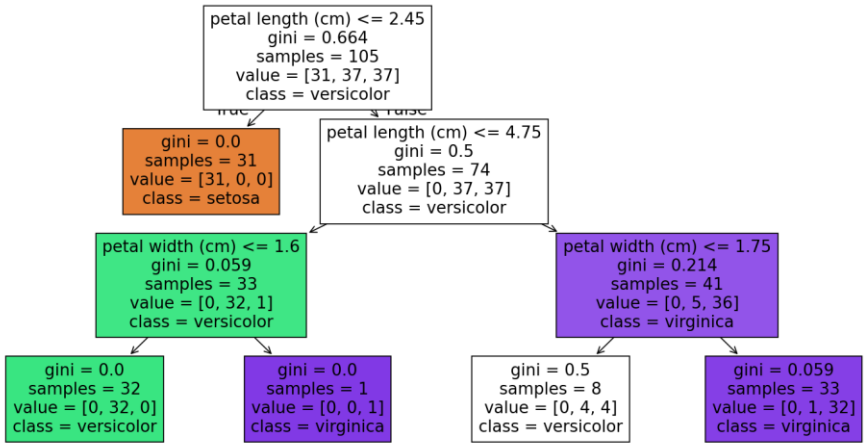
a) **Make predictions:**
Use the trained model to make predictions on the test set using the predict() method.
b) **Calculate evaluation metrics:**
Accuracy , Precision, Recall (Sensitivity) , F1-Score , Confusion Matrix

Step4: Stop

PROGRAM:

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
from sklearn.datasets import load_iris

# 1. Load a sample dataset (e.g., Iris dataset)
iris = load_iris()
X = iris.data
y = iris.target

# 2. Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# 3. Build the Decision Tree Classifier
# Initialize the classifier with desired parameters (e.g., max_depth to prevent overfitting)
clf = DecisionTreeClassifier(max_depth=3, random_state=42)

# Train the classifier on the training data
clf.fit(X_train, y_train)

# 4. Make predictions on the test set
y_pred = clf.predict(X_test)

5. Evaluate the model's performance
print("Decision Tree Classifier Evaluation:")
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=iris.target_names))

# Optional: Visualize the decision tree (requires graphviz and pydotplus)
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt
plt.figure(figsize=(15, 10))
plot_tree(clf, feature_names=iris.feature_names, class_names=iris.target_names,
filled=True)
plt.show()
```

OUTPUT:



Evaluation:

```
Decision Tree Classifier Evaluation:
Accuracy: 1.0000

Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        19
  versicolor       1.00      1.00      1.00        13
   virginica       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

Result: this program has been executes successfully and decision tree classification is performed.

Experiment :8(use google colab)

Problem statement:
Write a python program to implement a logistic regression model for binary classification on numerical dataset

Aim: To implement a logistic regression model for binary classification on numerical dataset

Algorithm:
Step1: Start
Step2: Data Preparation:

a) **Load and Inspect Data:** Load our numerical dataset and understand its structure, including features (independent variables) and the target variable (dependent variable) for binary classification.

b) **Handle Missing Values:** Address any missing values in our dataset through imputation or removal, depending on the data characteristics and impact.

c) **Feature Scaling:** Scale numerical features (e.g., using standardization or normalization) to ensure that features with larger ranges do not disproportionately influence the model's training. This is crucial for optimization algorithms like gradient descent.

d) **Split Data:** Divide the dataset into training and testing sets. The training set is used to train the model, and the testing set is used to evaluate its performance on unseen data. A common split is 70-80% for training and 20-30% for testing.

Step3: Model Definition:

Sigmoid Function: Define the sigmoid (or logistic) function, which maps any real-valued number to a value between 0 and 1. This function transforms the linear combination of features into a probability.

$\sigma(z) = 1 / (1 + e^{(-z)})$

where z is the linear combination of features and weights: $z = w0 + w1*x1 + w2*x2 + ... + wn*xn$.

Step4: Training the Model (Optimization):

a) **Initialize Weights:** Initialize the model's weights (coefficients) and bias (intercept) with small random values or zeros.

b) **Cost Function:** Define a cost function (e.g., binary cross-entropy or log-loss) to quantify the difference between predicted probabilities and actual binary labels. The goal is to minimize this cost.

c) **Gradient Descent:** Use an optimization algorithm like gradient descent to iteratively update the weights and bias. In each iteration:

- Calculate the predicted probabilities using the current weights and bias.
- Calculate the gradients of the cost function with respect to each weight and bias.
- Update the weights and bias in the direction that minimizes the cost, using a learning rate to control the step size.

d) **Iteration:** Repeat the gradient descent steps for a fixed number of iterations or until the cost function converges.

Step5: Prediction:

a) **Probability Prediction:** For new, unseen data, calculate the probability of the positive class using the trained weights, bias, and the sigmoid function.

b) **Class Assignment:** Classify the instance as belonging to the positive class if the predicted probability is above a certain threshold (commonly 0.5), and to the negative class otherwise.

Step6: Evaluation:

a) **Metrics:** Evaluate the model's performance on the testing set using appropriate metrics for binary classification, such as accuracy, precision, recall, F1-score, and ROC AUC.

b) **Confusion Matrix:** Generate a confusion matrix to visualize the model's predictions and identify true positives, true negatives, false positives, and false negatives.

Step7: Stop

PROGRAM:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.datasets import load_breast_cancer # Example numerical dataset
# Load a sample numerical dataset (e.g., Breast Cancer dataset)
data = load_breast_cancer()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Initialize the Logistic Regression model
model = LogisticRegression(solver='liblinear', random_state=42) # 'liblinear' is a good
choice for smaller datasets

# Train the model using the training data
model.fit(X_train, y_train)
# Make predictions on the test set
y_pred = model.predict(X_test)

# Evaluate the model's performance
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

OUTPUT:

```
Accuracy: 0.956140350877193

Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.91      0.94        43
           1       0.95      0.99      0.97        71

    accuracy                           0.96       114
   macro avg       0.96      0.95      0.95       114
weighted avg       0.96      0.96      0.96       114


Confusion Matrix:
 [[39  4]
 [ 1 70]]
```

Result: this program has been executed successfully and logistic regression model for binary classification is implemented on numerical dataset.

Experiment :9 (use google colab)

Problem statement:
Write a python program to apply PCA for feature reduction and visualization.

Aim: To apply PCA for feature reduction and visualization.

Algorithm:

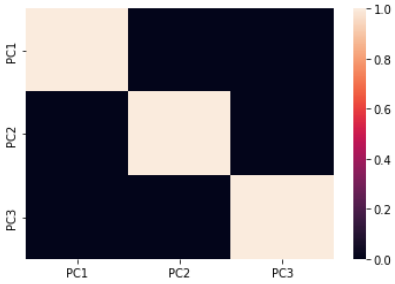
Step1: Start
Step2: Import necessary libraries
Step3: Load the dataset
Step4: Standardize the features
Step5: Applying Principal Component Analysis
Step6: Checking Co-relation between features after PCA
Step7: Stop

PROGRAM:

```
# Import necessary libraries
from sklearn import datasets  # to retrieve the iris Dataset
import pandas as pd  # to load the dataframe
from sklearn.preprocessing import StandardScaler  # to standardize the features
from sklearn.decomposition import PCA  # to apply PCA
import seaborn as sns  # to plot the heat maps
#Load the Dataset
iris = datasets.load_iris()
#convert the dataset into a pandas data frame
df = pd.DataFrame(iris['data'], columns = iris['feature_names'])
#display the head (first 5 rows) of the dataset
df.head()
#Standardize the features
#Create an object of StandardScaler which is present in sklearn.preprocessing
scalar = StandardScaler()
scaled_data = pd.DataFrame(scalar.fit_transform(df)) #scaling the data
scaled_data
#Applying PCA
#Taking no. of Principal Components as 3
pca = PCA(n_components = 3)
pca.fit(scaled_data)
data_pca = pca.transform(scaled_data)
data_pca = pd.DataFrame(data_pca,columns=['PC1','PC2','PC3'])
data_pca.head()
#Checking Co-relation between features after PCA
sns.heatmap(data_pca.corr())
```

OUTPUT:

	PC1	PC2	PC3
0	-2.264703	0.480027	0.127706
1	-2.080961	-0.674134	0.234609
2	-2.364229	-0.341908	-0.044201
3	-2.299384	-0.597395	-0.091290
4	-2.389842	0.646835	-0.015738



Result: this program has been executed successfully and Applied PCA for feature reduction and visualization on the dataset.

Experiment :10 (use google colab)

Problem statement:

Write a python program to design a simple neural network using TensorFlow/Keras for image classification.

Aim: To design a simple neural network using TensorFlow/Keras for image classification.

Algorithm:

Step1: Start
Step2: Import Libraries:
Import **tensorflow** , from tensorflow.keras, and other modules such as **Sequential** for model building, **Dense** for fully connected layers, and **datasets** for loading example data.
Step3: Load and Preprocess Data:
Load an image dataset, such as **MNIST**, and normalize the pixel values to a range between **0 and 1**. This helps in faster and more stable training.
Step4: Build the Model:
- Define the neural network architecture using **Sequential**. A common approach for image classification is to **flatten the input images** and then **add Dense layers**.
- The final Dense layer should have an output equal to the number of classes and use a **softmax** activation for multi-class classification.

Step5: Compile the Model:

- Configure the model for training by specifying an **optimizer** (e.g., 'adam'), a **loss function** (e.g., 'sparse_categorical_crossentropy' for integer labels), and **metrics** to monitor during training (e.g., 'accuracy').

Step6: Train the Model:

- Train the model using the training data (x_train, y_train) for a specified number of epochs.

Step7: Evaluate the Model:

Step7: Stop

PROGRAM:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
model = Sequential([
    Flatten(input_shape=(28, 28)), # Flattens the 28x28 image into a 784-element
vector
    Dense(128, activation='relu'), # Hidden layer with 128 neurons and ReLU activation
    Dense(10, activation='softmax') # Output layer for 10 classes (digits 0-9)
  ])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'\nTest accuracy: {test_acc}')
```

OUTPUT:

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.8749 - loss: 0.4433
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.9649 - loss: 0.1215
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9773 - loss: 0.0779
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.9832 - loss: 0.0550
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9868 - loss: 0.0435
313/313 - 1s - 3ms/step - accuracy: 0.9771 - loss: 0.0774

Test accuracy: 0.9771000146865845
```

Result: this program has been executed successfully and Design a simple neural network using TensorFlow/Keras for image classification.

Experiment :11 (use google colab)

Problem statement: Train and evaluate a CNN model for digit recognition using MNIST dataset.

Aim: to Train and evaluate a CNN model for digit recognition using MNIST dataset.

Algorithm:

Step1: Start
Step2: Load and preprocess the data:
The MNIST dataset can be loaded using the Keras library, and the images can be normalized to have pixel values between 0 and 1.
Step3: Define the model architecture:
The CNN can be constructed using the Keras **Sequential** API, which allows for easy building of sequential models layer-by-layer. The architecture should typically include convolutional layers, pooling layers, and fully-connected layers.
Step4: **Compile the model:**
The model needs to be compiled with a loss function, an optimizer, and a metric for evaluation.
Step5: Train the model:
The model can be trained on the training set using the Keras fit() function. It is important to monitor the training accuracy and loss to ensure the model is converging properly.
Step6: Evaluate the model:
The trained model can be evaluated on the test set using the Keras evaluate() function. The evaluation metric typically used for classification tasks is accuracy.
Step7: Stop.

PROGRAM:

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from keras.utils import to_categorical

# 1. Load and Preprocess the MNIST Dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Reshape data to include channel dimension (for CNN input)
# MNIST images are grayscale, so channel is 1
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)

# Convert pixel values to float32 and normalize to [0, 1]
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255

# Convert class vectors to binary class matrices (one-hot encoding)
num_classes = 10
y_train = to_categorical(y_train, num_classes)
```

```
y_test = to_categorical(y_test, num_classes)

# 2. Define the CNN Model Architecture
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

# 3. Compile the Model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# 4. Train the Model
batch_size = 128
epochs = 10
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, verbose=1,
validation_data=(x_test, y_test))

# 5. Evaluate the Model
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

OUTPUT:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_s
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 43s 88ms/step - accuracy: 0.7951 - loss: 0.6485 - val_accuracy: 0.9799 - val_loss: 0.0610
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 81s 86ms/step - accuracy: 0.9721 - loss: 0.0951 - val_accuracy: 0.9881 - val_loss: 0.0379
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 41s 86ms/step - accuracy: 0.9801 - loss: 0.0681 - val_accuracy: 0.9885 - val_loss: 0.0341
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 41s 86ms/step - accuracy: 0.9852 - loss: 0.0508 - val_accuracy: 0.9900 - val_loss: 0.0284
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 41s 86ms/step - accuracy: 0.9867 - loss: 0.0434 - val_accuracy: 0.9906 - val_loss: 0.0282
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 41s 86ms/step - accuracy: 0.9883 - loss: 0.0394 - val_accuracy: 0.9911 - val_loss: 0.0235
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 40s 83ms/step - accuracy: 0.9888 - loss: 0.0364 - val_accuracy: 0.9919 - val_loss: 0.0235
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 42s 86ms/step - accuracy: 0.9908 - loss: 0.0293 - val_accuracy: 0.9924 - val_loss: 0.0224
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 41s 86ms/step - accuracy: 0.9908 - loss: 0.0296 - val_accuracy: 0.9930 - val_loss: 0.0206
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 41s 86ms/step - accuracy: 0.9926 - loss: 0.0234 - val_accuracy: 0.9917 - val_loss: 0.0252
Test loss: 0.025154219940304756
Test accuracy: 0.9916999936103821
```

Result: this program has been executed successfully and Trained and evaluate a CNN model for digit recognition using MNIST dataset.

Experiment :12 (use notepad++ or any other IDE/IDLE)

Problem statement:
Write a python program for real time emotion recognition using webcam input and pre-trained model integration.

Aim: to implement real-time emotion recognition using webcam input and pre-trained model integration.

Algorithm:

Step1: Start
Step2: Installing OpenCV Library
pip install opencv-python
Step3: Importing OpenCV and Haar Cascade Classifier
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
Step4: **Open Webcam and Check for Webcam Access**
use the **cv2.VideoCapture()** function to open the webcam
Step5: Capture Frames, Convert to Grayscale and Detect Faces
- **gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)**
  Converts captured color frame (BGR) into grayscale for easier face detection.
- **faces=face_cascade.detectMultiScale(gray,scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))**
  Detects faces in the grayscale image by adjusting size, filtering false positives and setting the minimum face size.

Step6: Draw Rectangles Around Detected Faces and Display the Frame
Step7: **Exit the Program**
if cv2.waitKey(1) & 0xFF == ord('q')
step8: **Release the Webcam and Close All Windows**
When 'q' is pressed then webcam is released and any OpenCV windows are closed.
Step9: Stop.

PROGRAM:

```
import cv2
def main():

    face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')

    cap = cv2.VideoCapture(0)
    if not cap.isOpened():
        print("Error: Could not open webcam.")
        return

    while True:
        ret, frame = cap.read()
        if not ret:
```

```
            print("Error: Could not read frame.")
            break

        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5,
minSize=(30, 30))

        for (x, y, w, h) in faces:
            cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)

        cv2.imshow('Face Detection', frame)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()

if __name__ == "__main__":
    main()
```

OUTPUT:

Live face detection

Result: this program has been executed successfully and detected the real time emotion with webcam.