Introduction

Name - Atharv Sasturkar

Company Name - Infosys Springboard

Role - AI/ML Intern

Linked in -www.linkedin.com/in/atharv-sasturkar-62b035264

Phone no - 8208114633

INTRODUCTION

Breast cancer is one of the leading causes of cancer-related deaths among women globally. Early detection is critical in improving survival rates and treatment outcomes. Magnetic Resonance Imaging (MRI) has emerged as a highly effective tool for detecting and characterising breast tumours due to its superior imaging capabilities, providing detailed insights into soft tissue structures.

In this project, we focus on developing a predictive model for **Breast Cancer Prediction** using an **MRI dataset**. The goal is to design an automated system that accurately identifies and classifies breast tumours as either benign or malignant based on MRI scans. Leveraging the power of **Machine Learning (ML)** and **Convolutional Neural Networks (CNNs)**, the system aims to assist radiologists in making faster and more precise diagnoses.

DATA EXPLORATION

Data exploration is the first step of data analysis used to explore and visualize data to uncover insights from the start or identify areas or patterns to dig into more. Using interactive dashboards and point-and-click data exploration, users can better understand the bigger picture and get to insights faster.Data exploration is the initial step in data analysis and is used to understand a dataset before working with it.

METHODS:

Exploratory data analysis (EDA)

LOADING THE DATASET:

```
import os
import matplotlib.pyplot as plt
from PIL import Image
from torchvision import transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
import cv2


# Define the path to your dataset
data_dir = "breast_cancer_dataset"

# Check the directory structure
train_dir = os.path.join(data_dir, 'train')
val_dir = os.path.join(data_dir, 'val')
test_dir = os.path.join(data_dir, 'test')

# Print the classes available in the train directory
print("Classes in training set:")
print(os.listdir(train_dir))
```

```
print("\nClasses in val set:")
print(os.listdir(val_dir))

print("\nClasses in test set:")
print(os.listdir(test_dir))
```

Classes in training set:
['Benign', 'Malignant']

Classes in val set:
['Benign', 'Malignant']

Classes in test set:
['Benign', 'Malignant']

ANALYSING THE DATASET:

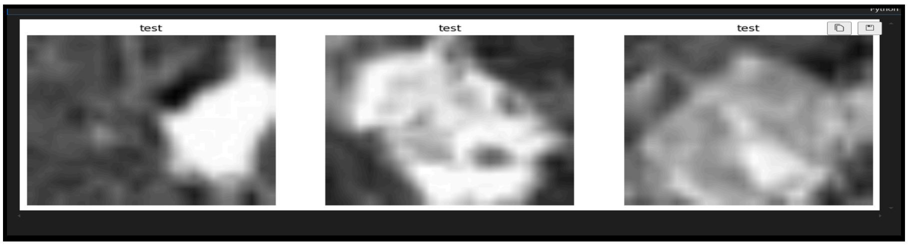
```
import matplotlib.pyplot as plt
from PIL import Image

def count_images(directory):

    class_counts = {}
    for class_name in os.listdir(directory):
        class_path = os.path.join(directory, class_name)
        if os.path.isdir(class_path):
            class_counts[class_name] = len(os.listdir(class_path))
    return class_counts

# Count images in the training set
train_class_counts = count_images(train_dir)
print("Number of images in each class (train):", train_class_counts)
```

Number of images in each class (train): {'Benign': 61, 'Malignant': 105}

DISPLAYING SAMPLE OF IMAGES :

```
import os
import random
import torch
import torchvision.transforms as transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
```

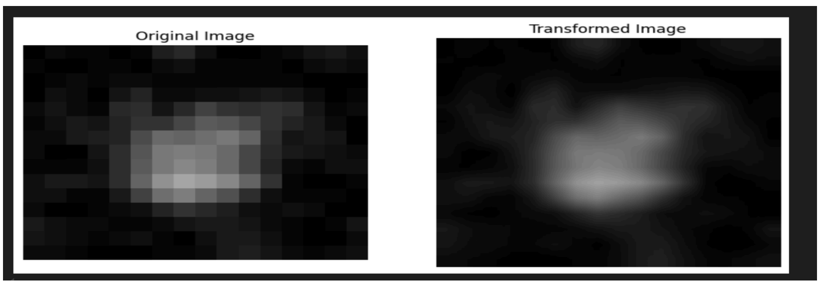
```
import matplotlib.pyplot as plt

# Define the path to your dataset
data_dir = 'breast_cancer_dataset'  # Use a raw string for Windows path

# Define the transformation for the images
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize images to 224x224
    transforms.ToTensor(),            # Convert images to PyTorch
tensors
])

# Load the dataset
dataset = ImageFolder(root=data_dir, transform=transform)
class_names = dataset.classes  # Get class names

# Create a DataLoader
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)

def display_random_images(dataset, class_names, num_images=3):
    # Get a random sample of images for each class
    for class_index, class_name in enumerate(class_names):
        # Filter images belonging to the current class
        class_images = [dataset[i][0] for i in range(len(dataset)) if
dataset[i][1] == class_index]

        # Randomly select a few images
        if len(class_images) > 0:
            selected_images = random.sample(class_images,
min(num_images, len(class_images)))
            plt.figure(figsize=(15, 5))
            for i in range(len(selected_images)):
                plt.subplot(1, len(selected_images), i + 1)
                plt.imshow(selected_images[i].permute(1, 2, 0).numpy())
# Convert from CxHxW to HxWxC
                plt.title(class_name)
                plt.axis('off')
            plt.show()

# Display random images from each class
display_random_images(dataset, class_names)
```



DATA TRANSFORMATION:

```
import torchvision.transforms as transforms
import torch

# Define transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resizing to 224x224 pixels
    transforms.ToTensor(),  # Converting to PyTorch tensor

    # Normalization- process of scaling pixel intensity values to a common range
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # Normalize with ImageNet values
])
import torch
from PIL import Image
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

def show_image_transformation(image_path):
    """
    Load an image from a path, apply transformation, and display both original and transformed images.

    Args:
        image_path (str): Path to the image file.
    """
    # Load the image
    image = Image.open(image_path).convert("RGB")

    # Define the transformation (example: resize, convert to tensor, normalize)
    transform = transforms.Compose([
        transforms.Resize((128, 128)),  # Resize to 128x128
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
    ])
```

```
    # Apply the transformation
    transformed_image = transform(image)

    # Unnormalize for visualization
    unnormalize = transforms.Normalize(
        mean=[-0.5 / 0.5, -0.5 / 0.5, -0.5 / 0.5],
        std=[1 / 0.5, 1 / 0.5, 1 / 0.5]
    )
    unnormalized_image = unnormalize(transformed_image)

    # Convert tensor to numpy array for visualization
    original_image_np = transforms.ToTensor()(image).permute(1, 2, 0).numpy()
    transformed_image_np = unnormalized_image.permute(1, 2, 0).numpy().clip(0, 1)

    # Plot both images
    fig, axs = plt.subplots(1, 2, figsize=(10, 5))
    axs[0].imshow(original_image_np)
    axs[0].set_title("Original Image")
    axs[0].axis("off")

    axs[1].imshow(transformed_image_np)
    axs[1].set_title("Transformed Image")
    axs[1].axis("off")

    plt.show()

# Usage example
image_path = input("test\Benign\BreaDM-Be-1810\SUB1\p-030.jpg ")
show_image_transformation(image_path)
```



DATA AUGMENTATION:

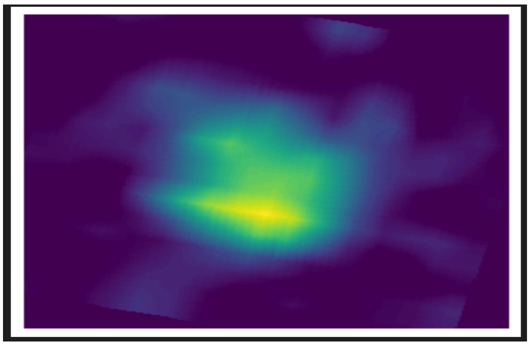
Data augmentation is a machine learning technique that can improve the performance of convolutional neural networks (CNNs) by increasing the size and diversity of training data.Data augmentation has been widely implemented in research for a range of computer vision tasks, from image classification to object detection. As such, there is a wealth of research on how augmented images improve the performance of state-of-the-art convolutional neural networks (CNNs) in image processing.Data augmentation is often used when data is imbalanced, but it can also be used to make data imbalanced to bias a model towards a certain case. However, augmentation can lead to overfitting in cases with very few data samples.

```
# Define data augmentation with additional techniques
augmentation_transforms = transforms.Compose([
   transforms.Resize((224, 224)),
   transforms.RandomHorizontalFlip(),  # Randomly flip the image horizontally
   transforms.RandomRotation(10),  # Random rotation
   transforms.RandomCrop(224, padding=4),  # Random crop with padding
   transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.2),  # Random
color jitter
   transforms.RandomGrayscale(p=0.1),  # Randomly convert image to grayscale with a
probability of 0.1
   transforms.RandomPerspective(distortion_scale=0.5, p=0.5),  # Random perspective
transformation
   transforms.ToTensor(),
   transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

TECHNIQUES FOR DATA AUGMENTATION:
Mirroring and random cropping =>methods of data augmentation(main thing have to do)Augmenting images by colour shifting
VISUALISING AUGMENTED IMAGE:

```
import torch
import torchvision.transforms as transforms
from PIL import Image
import matplotlib.pyplot as plt

# Load an example image (replace 'path_to_image.jpg' with the path to
your image)
image_path = 'test\Benign\BreaDM-Be-1810\SUB1\p-030.jpg '
image = Image.open(image_path)

# Define the augmentation transforms
transform = transforms.Compose([
    transforms.Resize((256, 256)),          # Resize to 256x256
```

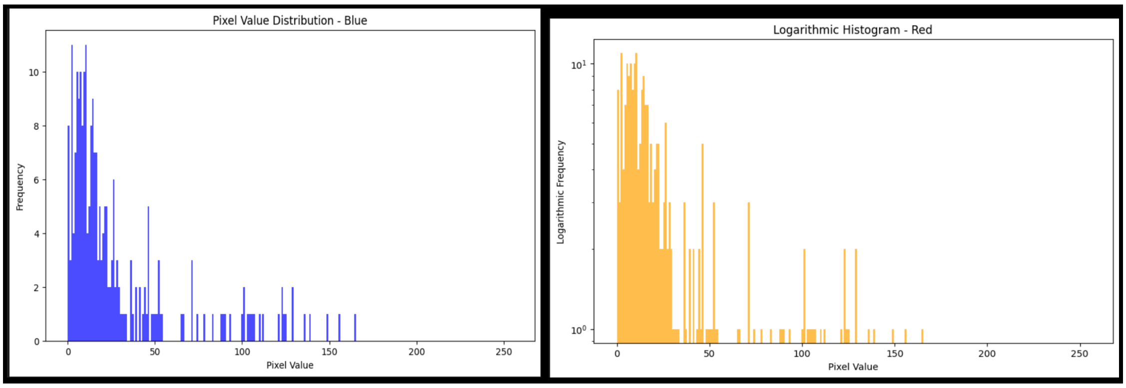
```
    transforms.RandomHorizontalFlip(),        # Randomly flip image
horizontally
    transforms.RandomRotation(30),            # Randomly rotate by up to
30 degrees
    transforms.ColorJitter(brightness=0.5, contrast=0.5,
saturation=0.5), # Random brightness, contrast, saturation
    transforms.RandomAffine(15),              # Random affine
transformation
])

# Apply the transformations to the image
augmented_image = transform(image)

# Convert the image to a format suitable for displaying
plt.imshow(augmented_image)
plt.axis('off')  # Turn off axis
plt.show()
```



Histogram of pixel values:

```
import os
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

# Image path
image_path ='test\Benign\BreaDM-Be-1810\SUB1\p-030.jpg'

# Check if the file exists
if not os.path.exists(image_path):
    print(f"Error: The file at {image_path} does not exist.")
else:
    # Load the image and convert it to grayscale
    try:
```

```
        image = Image.open(image_path).convert('L')

        # Convert the image to a NumPy array
        image_array = np.array(image)

        # Flatten the image array to 1D
        image_flat = image_array.flatten()

        # Plot basic histogram with color
        plt.figure(figsize=(10, 6))
        plt.hist(image_flat, bins=256, range=(0, 255), color='blue', alpha=0.7)
        plt.title('Pixel Value Distribution - Blue')
        plt.xlabel('Pixel Value')
        plt.ylabel('Frequency')
        plt.show()

        # Cumulative Histogram
        plt.figure(figsize=(10, 6))
        plt.hist(image_flat, bins=256, range=(0, 255), color='green', alpha=0.7, cumulative=True)
        plt.title('Cumulative Histogram - Green')
        plt.xlabel('Pixel Value')
        plt.ylabel('Cumulative Frequency')
        plt.show()

        # Logarithmic Histogram
        plt.figure(figsize=(10, 6))
        plt.hist(image_flat, bins=256, range=(0, 255), color='orange', alpha=0.7, log=True)
        plt.title('Logarithmic Histogram - Red')
        plt.xlabel('Pixel Value')
        plt.ylabel('Logarithmic Frequency')
        plt.show()

    except OSError as e:
        print(f"Error opening the image: {e}")
```

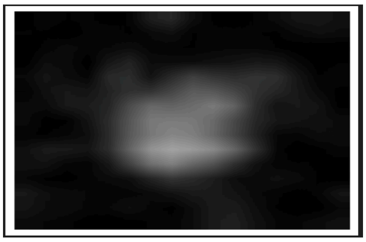


SETTING UP DATALOADERS:

Creating Dataset with updated transformation:

```
import torch
from torchvision import transforms
from torch.utils.data import DataLoader, Dataset
from PIL import Image
import matplotlib.pyplot as plt

# Define the transformation for a single image (resize to 224x224)
transform = transforms.Compose([
    transforms.Resize((224, 224)),       # Resize to 224x224
    transforms.ToTensor(),               # Convert to tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225])  # Normalize
])

# Define a custom dataset for a single image
class SingleImageDataset(Dataset):
    def __init__(self, image_path, transform=None):
        self.image_path = image_path
        self.transform = transform

    def __len__(self):
        return 1  # Only one image

    def __getitem__(self, idx):
        image = Image.open(self.image_path).convert("RGB")  # Load and
convert image to RGB
        if self.transform:
            image = self.transform(image)
        return image

# Load a single image and apply transformations
```

```
image_path ='test\Benign\BreaDM-Be-1810\SUB1\p-030.jpg'  # Replace with
the path to your image
dataset = SingleImageDataset(image_path=image_path, transform=transform)

# Create a DataLoader for the single image dataset
dataloader = DataLoader(dataset, batch_size=1, shuffle=False)

# Display the transformed image
for img in dataloader:
    img = img.squeeze(0)  # Remove the batch dimension
    img = img.permute(1, 2, 0)  # Rearrange dimensions for displaying

    # Unnormalize for display
    img = img * torch.tensor([0.229, 0.224, 0.225]) +
torch.tensor([0.485, 0.456, 0.406])
    img = img.clamp(0, 1)  # Clamp values to range [0, 1]

    plt.imshow(img)
    plt.axis('off')
    plt.show()
    break
```

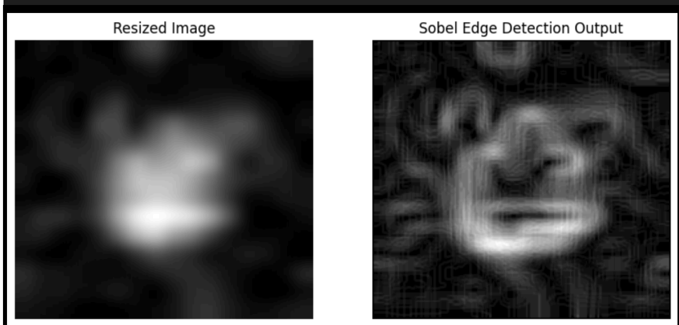


SOBEL OPERATOR:(resizing the image)

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

# Define Sobel kernels
sobel_x = np.array([[-1, 0, 1],
                    [-2, 0, 2],
                    [-1, 0, 1]])

sobel_y = np.array([[-1, -2, -1],
                    [ 0,  0,  0],
                    [ 1,  2,  1]])
```

```python
# Function to apply convolution between image and kernel
def apply_convolution(image, kernel):
    height, width = image.shape
    output_image = np.zeros((height, width))

    # Apply convolution
    for i in range(1, height-1):
        for j in range(1, width-1):
            region = image[i-1:i+2, j-1:j+2]  # 3x3 region around the pixel
            output_image[i, j] = np.sum(region * kernel)

    return output_image

# Function to compute gradient magnitude from Sobel outputs
def compute_gradient_magnitude(sobel_x_output, sobel_y_output):
    return np.sqrt(sobel_x_output**2 + sobel_y_output**2)

# Function to normalize the image for display
def normalize_image(image):
    image_min = np.min(image)
    image_max = np.max(image)
    normalized_image = (image - image_min) / (image_max - image_min) * 255
    return normalized_image.astype(np.uint8)

# Load your image (replace 'your_image.jpg' with the path to your image file)
input_image_path = 'test\Benign\BreaDM-Be-1810\SUB1\p-030.jpg'  # Change this to your image path
image = Image.open(input_image_path).convert('L')  # Convert to grayscale

# Resize the image to a width of 225 pixels
new_width = 225
aspect_ratio = image.height / image.width
new_height = int(new_width * aspect_ratio)
image_resized = image.resize((new_width, new_height))

# Convert the resized image to a NumPy array
image_array = np.array(image_resized)

# Apply Sobel operators
```

```python
sobel_x_output = apply_convolution(image_array, sobel_x)
sobel_y_output = apply_convolution(image_array, sobel_y)

# Calculate gradient magnitude (Sobel magnitude)
sobel_magnitude = compute_gradient_magnitude(sobel_x_output,
sobel_y_output)

# Normalize the output image for better visualization
normalized_output = normalize_image(sobel_magnitude)

# Convert to PIL Image
output_image = Image.fromarray(normalized_output)

# Save the output image
output_image.save('sobel_edge_detection_output.png')
print("Sobel edge detection output saved as
'sobel_edge_detection_output.png'")

# Display the original resized and the output image using matplotlib
plt.figure(figsize=(10, 5))

# Display original resized image
plt.subplot(1, 2, 1)
plt.title("Resized Image")
plt.imshow(image_array, cmap='gray')
plt.axis('off')  # Hide axes

# Display Sobel output image
plt.subplot(1, 2, 2)
plt.title("Sobel Edge Detection Output")
plt.imshow(normalized_output, cmap='gray')
plt.axis('off')  # Hide axes

plt.show()  # Show the figure with both images
```



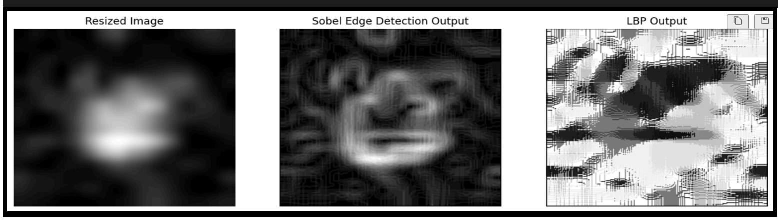

```
sobel edge detection output
 [[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   2   7  11  13  17  49  58  40   4  13   3  21  44  48   0]
 [  0  23  17  11  38  44  22   6  14  25  30  39  34  10  20   0]
 [  0  30  23  31  80  62  57  82 108  99  86  89  89  70  25   0]
 [  0  23  33  55  80  46  78 141 153 146 105  62  65  79  33   0]
 [  0  31  20  43  79  87 143 148 121 134 123  75  62  52  46   0]
 [  0  32  44  42 106 147 136 114  74  60 138 129  50  19  41   0]
 [  0  20  46  73 136 160  86  55  37  92 155 131  53  28  18   0]
 [  0  42  43  72 157 179 117  75  92 134 164 142  71  38  31   0]
 [  0  22   7  53 160 201 129  26  80 127 169 158  78  10  21   0]
 [  0  44  48  56 141 233 255 237 248 244 213 126  41  22   6   0]
 [  0  21   7  16  61 146 211 211 175 130  88  34  12  15  11   0]
 [  0  33  19   6   5  24  62  87  60  18  37  27  25  22  20   0]
 [  0  24  19  19  16  14  17  30  47  21  31  48  30  12  27   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]]
Sobel edge detection output saved as 'sobel_edge_detection_output.png'
```

LOCAL BINARY PATTERN:

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
# Define Sobel kernels
sobel_x = np.array([[-1, 0, 1],
                    [-2, 0, 2],
                    [-1, 0, 1]])
sobel_y = np.array([[-1, -2, -1],
                    [ 0,  0,  0],
                    [ 1,  2,  1]])
# Function to apply convolution between image and kernel
def apply_convolution(image, kernel):
    height, width = image.shape
    output_image = np.zeros((height, width))
      # Apply convolution
    for i in range(1, height-1):
        for j in range(1, width-1):
            region = image[i-1:i+2, j-1:j+2]  # 3x3 region around the
pixel
            output_image[i, j] = np.sum(region * kernel)

    return output_image

# Function to compute gradient magnitude from Sobel outputs
def compute_gradient_magnitude(sobel_x_output, sobel_y_output):
    return np.sqrt(sobel_x_output**2 + sobel_y_output**2)
# Function to normalize the image for display
def normalize_image(image):
    image_min = np.min(image)
    image_max = np.max(image)
    normalized_image = (image - image_min) / (image_max - image_min) *
255
    return normalized_image.astype(np.uint8)
# Function to compute Local Binary Pattern (LBP)
def compute_lbp(image):
    height, width = image.shape
    lbp_image = np.zeros((height, width), dtype=np.uint8)

    for i in range(1, height-1):
        for j in range(1, width-1):
```

```
            center_pixel = image[i, j]
            binary_string = ''
# Clockwise comparison of the 8 neighbors starting from the top-left
            binary_string += '1' if image[i-1, j-1] >= center_pixel else
'0'
            binary_string += '1' if image[i-1, j] >= center_pixel else
'0'
            binary_string += '1' if image[i-1, j+1] >= center_pixel else
'0'
            binary_string += '1' if image[i, j+1] >= center_pixel else
'0'
            binary_string += '1' if image[i+1, j+1] >= center_pixel else
'0'
            binary_string += '1' if image[i+1, j] >= center_pixel else
'0'
            binary_string += '1' if image[i+1, j-1] >= center_pixel else
'0'
            binary_string += '1' if image[i, j-1] >= center_pixel else
'0'

# Convert binary string to decimal and assign to the LBP image
            lbp_image[i, j] = int(binary_string, 2)

    return lbp_image
# Load your image
input_image_path = 'test\Benign\BreaDM-Be-1810\SUB1\p-030.jpg'  # Change
this to your image path
image = Image.open(input_image_path).convert('L')  # Convert to
grayscale
# Resize the image to a width of 225 pixels
new_width = 225
aspect_ratio = image.height / image.width
new_height = int(new_width * aspect_ratio)
image_resized = image.resize((new_width, new_height))
# Convert the resized image to a NumPy array
image_array = np.array(image_resized)
# Apply Sobel operators
sobel_x_output = apply_convolution(image_array, sobel_x)
sobel_y_output = apply_convolution(image_array, sobel_y)
# Calculate gradient magnitude (Sobel magnitude)
sobel_magnitude = compute_gradient_magnitude(sobel_x_output,
sobel_y_output)
# Normalize the Sobel output image for better visualization
```

```python
normalized_sobel_output = normalize_image(sobel_magnitude)
# Compute Local Binary Pattern (LBP)
lbp_image = compute_lbp(image_array)
# Normalize the LBP output for better visualization
normalized_lbp_output = normalize_image(lbp_image)
# Convert to PIL Image
sobel_output_image = Image.fromarray(normalized_sobel_output)
lbp_output_image = Image.fromarray(normalized_lbp_output)
# Save the Sobel and LBP output images
sobel_output_image.save('sobel_edge_detection_output.png')
lbp_output_image.save('lbp_output.png')
print("Sobel and LBP outputs saved as 'sobel_edge_detection_output.png'
and 'lbp_output.png'")
# Display the original resized, Sobel output, and LBP output images
using matplotlib
plt.figure(figsize=(15, 5))
# Display original resized image
plt.subplot(1, 3, 1)
plt.title("Resized Image")
plt.imshow(image_array, cmap='gray')
plt.axis('off')  # Hide axes
# Display Sobel output image
plt.subplot(1, 3, 2)
plt.title("Sobel Edge Detection Output")
plt.imshow(normalized_sobel_output, cmap='gray')
plt.axis('off')  # Hide axes
# Display LBP output image
plt.subplot(1, 3, 3)
plt.title("LBP Output")
plt.imshow(normalized_lbp_output, cmap='gray')
plt.axis('off')  # Hide axes

plt.show()  # Show the figure with all images
```



MVM-LBP IMAGES ON 5 IMAGES:

LBP Mean based, LBP Variance based, LBP Median based, MVM Thresholded:

```python
import numpy as np
from PIL import Image
```

```
import matplotlib.pyplot as plt
from scipy.ndimage import generic_filter
import os
# Function to normalize the image for display
def normalize_image(image):
    image_min = np.min(image)
    image_max = np.max(image)
    normalized_image = ((image - image_min) / (image_max - image_min) *
255).astype(np.uint8)
    return normalized_image
# Function to compute Local Binary Pattern (LBP)
def compute_lbp(image, radius=1, neighbors=8):
    height, width = image.shape
    lbp_image = np.zeros((height, width), dtype=np.uint8)
        for i in range(radius, height - radius):
         for j in range(radius, width - radius):
            center_pixel = image[i, j]
            binary_string = ''
            offsets = [(dy, dx) for dy in range(-radius, radius+1) for
dx in range(-radius, radius+1) if dy != 0 or dx != 0]
            for dy, dx in offsets[:neighbors]:  # Take only specified
neighbors
                binary_string += '1' if image[i + dy, j + dx] >=
center_pixel else '0'
            lbp_image[i, j] = int(binary_string, 2)
        return lbp_image
# Define functions to compute local mean, variance, and median
def local_mean(image, size=3):
    return generic_filter(image, np.mean, size=size)
def local_variance(image, size=3):
    return generic_filter(image, np.var, size=size)
def local_median(image, size=3):
    return generic_filter(image, np.median, size=size)
# Apply MVM threshold using the formula
def apply_mvm_threshold(mean_image, variance_image, median_image):
    threshold = (mean_image + np.sqrt(variance_image) + median_image) /
3
    binary_image = (mean_image > threshold) * 255  # Apply threshold to
the mean image (or change as desired)
    return binary_image.astype(np.uint8)
# List of image paths (replace these with paths to your images)
image_paths = [
    'test/Benign/BreaDM-Be-1810/SUB1/p-030.jpg',
```

```
    'test/Benign/BreaDM-Be-1810/SUB2/p-030.jpg',
    'test/Benign/BreaDM-Be-1810/SUB3/p-030.jpg',
    'test/Benign/BreaDM-Be-1810/SUB5/p-030.jpg',
    'test/Benign/BreaDM-Be-1810/SUB6/p-030.jpg'
]
# Directory to save output images
output_dir = "output_images"
os.makedirs(output_dir, exist_ok=True)
# Process each image in the list
for idx, image_path in enumerate(image_paths, start=1):
    # Load and resize image using Lanczos filter for high quality
    image = Image.open(image_path).convert('L')
    new_width = 225
    aspect_ratio = image.height / image.width
    new_height = int(new_width * aspect_ratio)
    image_resized = image.resize((new_width, new_height), Image.LANCZOS)
    image_array = np.array(image_resized)
   # Compute LBP image
    lbp_image = compute_lbp(image_array, radius=1, neighbors=8)
    # Apply mean, variance, and median transformations to LBP image
    lbp_mean_based = local_mean(lbp_image, size=3)
    lbp_variance_based = local_variance(lbp_image, size=3)
    lbp_median_based = local_median(lbp_image, size=3)
   # Normalize for better display
    lbp_mean_based = normalize_image(lbp_mean_based)
    lbp_variance_based = normalize_image(lbp_variance_based)
    lbp_median_based = normalize_image(lbp_median_based)
   # Apply MVM threshold using the formula
    mvm_thresholded = apply_mvm_threshold(lbp_mean_based,
lbp_variance_based, lbp_median_based)
    # Display the images
    plt.figure(figsize=(20, 10))

    # Display original resized image
    plt.subplot(2, 5, 1)
    plt.title(f"Original Image {idx}")
    plt.imshow(image_array, cmap='gray')
    plt.axis('off')
    # Display LBP image
    plt.subplot(2, 5, 2)
    plt.title(f"LBP Image {idx}")
    plt.imshow(lbp_image, cmap='gray')
    plt.axis('off')
```

```
    # Display LBP mean-based image
    plt.subplot(2, 5, 3)
    plt.title(f"LBP Mean-based Image {idx}")
    plt.imshow(lbp_mean_based, cmap='gray')
    plt.axis('off')
    # Display LBP variance-based image
    plt.subplot(2, 5, 4)
    plt.title(f"LBP Variance-based Image {idx}")
    plt.imshow(lbp_variance_based, cmap='gray')
    plt.axis('off')
    # Display LBP median-based image
    plt.subplot(2, 5, 5)
    plt.title(f"LBP Median-based Image {idx}")
    plt.imshow(lbp_median_based, cmap='gray')
    plt.axis('off')
    # Display MVM thresholded image
    plt.subplot(2, 5, 6)
    plt.title("MVM Thresholded")
    plt.imshow(mvm_thresholded, cmap='gray')
    plt.axis('off')
    plt.tight_layout()
    plt.show()
```