PROJECT TITLE:

TumorTrace: MRI-Based AI for Breast Cancer Detection

INTRODUCTION:

Breast cancer is one of the most common cancers among women, making early detection essential for improving treatment outcomes. Magnetic Resonance Imaging (MRI) is a valuable diagnostic tool that offers detailed images of breast tissue, allowing for the detection of tumors that may not be visible through traditional methods. However, the interpretation of MRI scans can be subjective, leading to potential diagnostic inaccuracies.

The TumorTrace project aims to enhance breast cancer detection by leveraging artificial intelligence (AI) and machine learning (ML) techniques. By automating the identification of malignant and benign tumors in MRI scans, TumorTrace seeks to support healthcare professionals in making more accurate and timely diagnoses. Ultimately, this project aspires to improve early detection and patient outcomes in breast cancer care.

DATASET:

The dataset selected for the TumorTrace project consists of MRI images that have undergone image segmentation, ensuring precise delineation of regions of interest (ROIs) related to breast tumors. This segmentation process enhances the quality of the images by highlighting relevant features while reducing background noise, thereby facilitating more accurate analysis. The dataset is divided into three distinct subsets: training, testing, and validation.

WHY HAS THE DATASET BEEN RESIZED AND NORMALIZED AS PER IMAGENET DATASET DIMENSIONS?

Resizing and normalizing dataset as per ImageNet dimensions is a standard practice in deep learning that enhances model compatibility, improves performance through transfer learning, and ensures consistent training and evaluation conditions. It's a way to harness the power of models pretrained on large datasets to solve specific problems more effectively and efficiently.

CONVOLUTIONAL NEURAL NETWORKS:

The CS231n course overview on convolutional networks (CNNs/ConvNets):

- **Neural Networks and Image Data**: ConvNets are a specialized type of neural network that excel at handling grid-like data, such as images. Instead of fully connecting each neuron to every input, ConvNets use a different architecture that exploits spatial hierarchies in data.
- **Layers in ConvNets**:

    - **Convolutional Layer**: This layer applies a set of filters to the input image, creating feature maps that detect various elements (e.g., edges, textures). Each filter slides across the image and computes dot products between the filter values and the input.
    - **Pooling Layer**: This layer down-samples the input (usually after the convolutional layer) to reduce its dimensions and computational complexity. Max pooling is a common technique that selects the maximum value from a region of the image.
    - **Fully Connected Layer**: After multiple convolution and pooling layers, the high-level reasoning in the network is typically done by fully connected layers that aggregate the features learned and classify the data.

- **Benefits:** Their architecture allows ConvNets to capture local dependencies, preserve spatial hierarchies, and reduce the number of parameters (compared to fully connected networks).

- **Architecture:**
    - CNNs are composed of layers: Convolutional (CONV), Pooling (POOL), and Fully Connected (FC)
    - Input layers are typically divisible by 2 many times (e.g., 32, 64, 224)
- **Convolutional Layer:**
    - Uses learnable filters to detect features
    - Hyperparameters: filter size (F), stride (S), padding (P)
    - Typically use small filters (3x3 or 5x5) with stride 1 and padding
- **Pooling Layer:**
    - Reduces spatial dimensions of input
    - Common: Max pooling with 2x2 receptive field and stride 2
    - No learnable parameters
- **Fully Connected Layer:**
    - Similar to traditional neural networks
    - Can be converted to CONV layers and vice versa
- **Design Principles:**
    - Prefer stacking small filters over using large receptive fields
    - Use padding to maintain spatial dimensions in CONV layers
    - Let POOL layers handle downsampling
- **Famous Architectures:**
    - LeNet, AlexNet, ZFNet, GoogLeNet, VGGNet, ResNet
- **Computational Considerations:**
    - Memory is often the main bottleneck
    - Sources of memory usage: activations, parameters, and miscellaneous data
    - Reducing batch size can help fit models in memory
- **Parameter Sharing:**
    - Reduces number of parameters
    - Allows detection of features regardless of position in image
- **Implementation:**
    - Can be implemented as matrix multiplication using im2col operation
    - Backpropagation for CONV is also a convolution operation

WHY CNNS ARE MORE EFFICIENT THAN FULLY CONNECTED NETWORKS FOR IMAGE CLASSIFICATION?

Convolutional Neural Networks (CNNs) are more efficient than fully connected networks (FCNs) for image classification due to their ability to exploit the spatial structure of images. Here's why:

a. Spatial Hierarchies:

- **Local connectivity**: CNNs use convolutional layers that apply filters to small regions of the image, capturing spatial features like edges, textures, and shapes. This localized approach helps capture the hierarchical structure of images (from low-level features to high-level features).
- **Shared weights**: CNNs use the same filter (kernel) across the image, which means fewer parameters compared to FCNs, where every pixel is connected to every node in the next layer. This reduces the complexity and the risk of overfitting, making CNNs more efficient for images.
- **Translation invariance**: CNNs can detect patterns anywhere in the image because filters slide over the image, allowing for translation invariance, meaning a feature can be recognized no matter where it appears.

b. Parameter Efficiency:

- **Fewer parameters**: In FCNs, every input pixel is connected to every neuron in the next layer, leading to an enormous number of parameters, especially for large images. In CNNs, the use of convolutional filters significantly reduces the number of connections and parameters.
- **Pooling layers**: CNNs use pooling (usually max-pooling) to reduce the spatial size of the image representations, which further reduces the number of parameters and speeds up computation, all while retaining important features.

c. Feature Extraction:

CNNs automatically learn relevant features from the data through multiple layers of convolutions, while FCNs do not have the spatial awareness to capture hierarchical and local image features efficiently.

DATA AUGMENTATION:

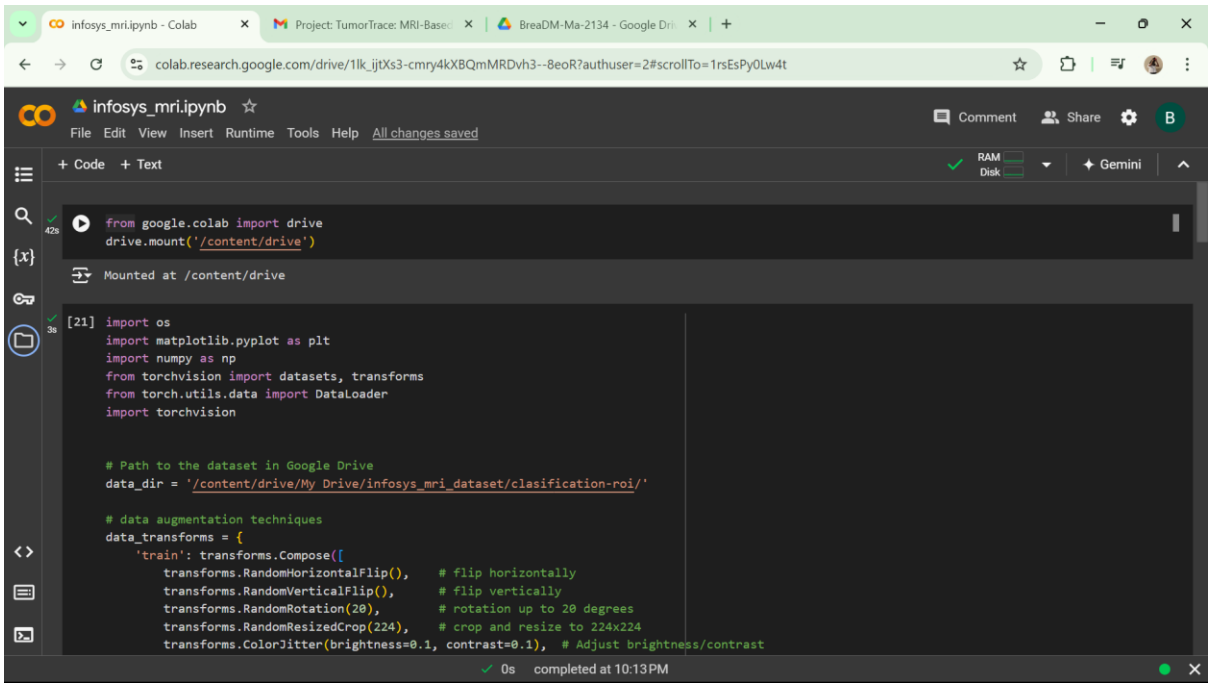
Data augmentation is a technique used in machine learning to increase the size and diversity of a training dataset by creating modified versions of existing data points. This helps improve the model's robustness and generalization capabilities.

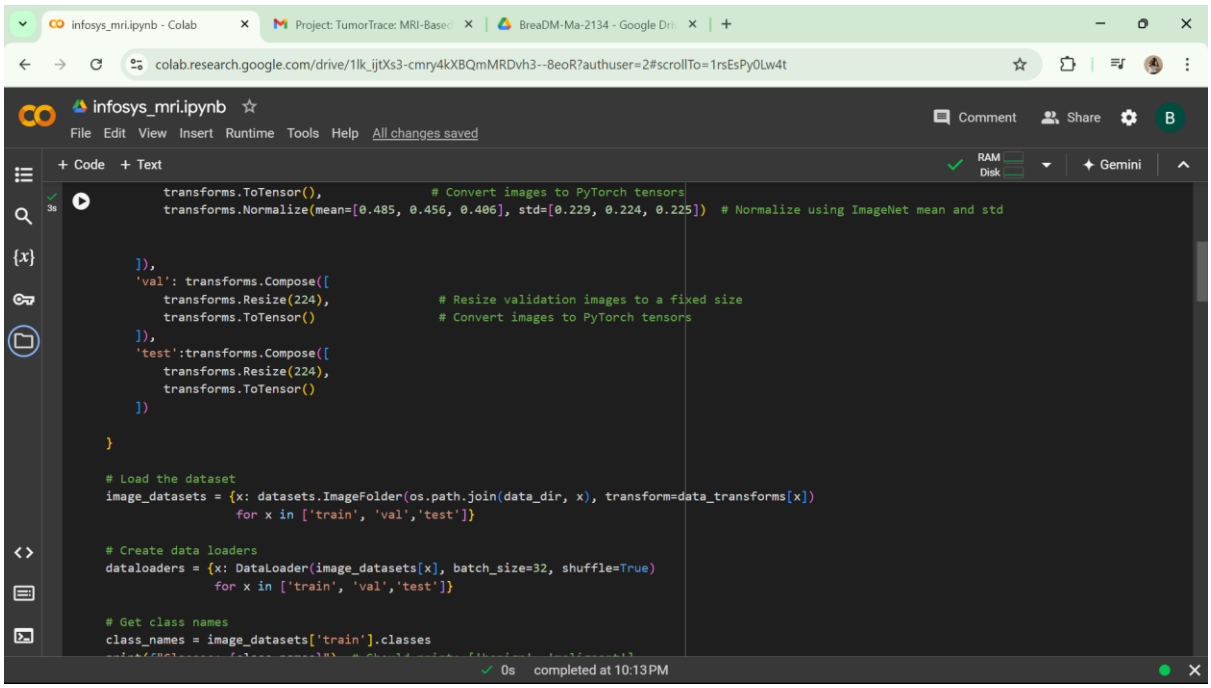
Key Points:

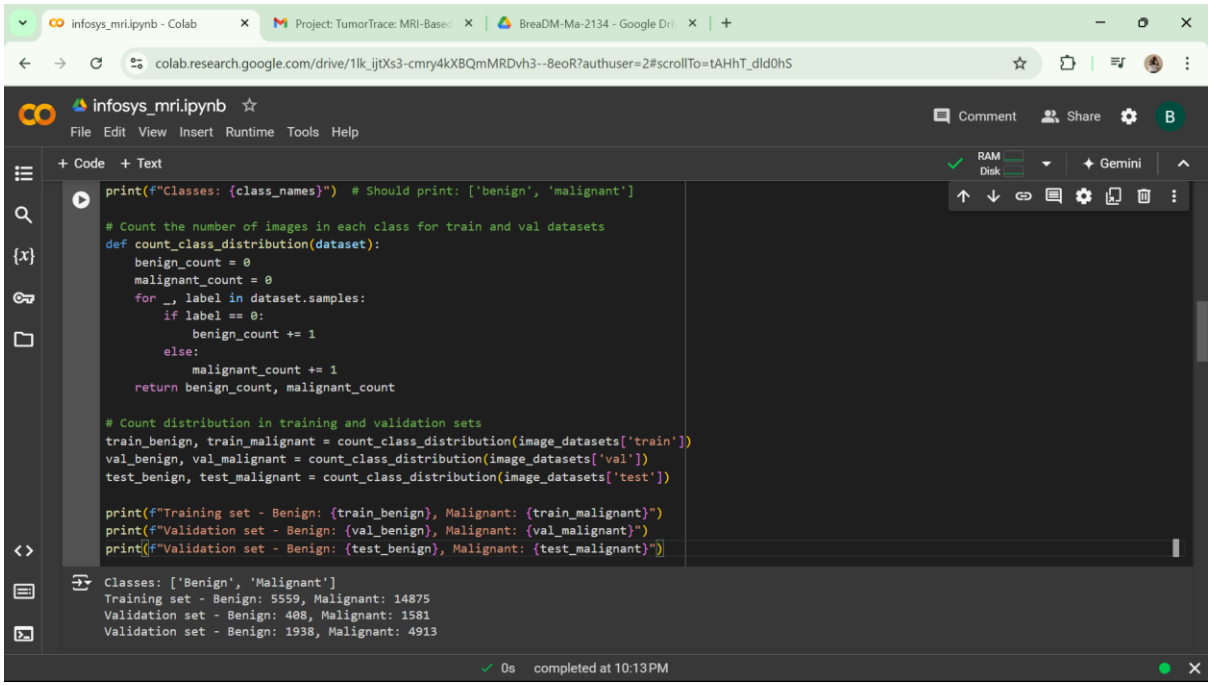
1. **Purpose:**
    - To artificially expand the dataset, reducing overfitting and enhancing model performance on unseen data.
2. **Common Techniques:**
    - Geometric Transformations: Rotation, flipping, translation, scaling.
    - Color Adjustments: Brightness, contrast, saturation changes.
    - Distortions: Shearing, zooming, cropping.

- Noise Addition: Introducing random noise to make models more robust.

3. **Implementation:**
   - Can be done on-the-fly during training or as a pre-processing step.
4. **Benefits:**
   - Improves accuracy, reduces overfitting, and is a cost-effective way to enhance datasets without needing more labeled data.







WHAT IS FEATURE EXTRACTION?

Feature extraction refers to the process of transforming raw data into a set of features that better represent the underlying patterns in the data, making it easier for a model to classify or make predictions. In the context of CNNs and image classification:

- **Initial Layers**: CNNs extract low-level features like edges, corners, and textures.
- **Deeper Layers**: As you move deeper into the network, CNNs extract more complex features, such as shapes, objects, or even entire faces, in image classification tasks.
- **Importance**: Feature extraction is crucial because it allows the network to reduce the input data to more meaningful and manageable representations, leading to better learning and performance.

In traditional machine learning, feature extraction was done manually by selecting specific characteristics, but CNNs automate this process.

VGG-16 ARCHITECTURE:

VGG-16 consists of:

- 13 Convolutional layers
- 3 Fully connected layers

The architecture can be summarized as:

1. **Convolutional Layers**:
   - 2 Conv layers: 64 filters (3×3 kernels), input 224×224×3 → output 224×224×64 (conv1_1 and conv1_2)
   - 2 Conv layers: 128 filters (3×3 kernels), input 112×112×64 → output 112×112×128 (conv2_1 and conv2_2)

- 3 Conv layers: 256 filters (3×3 kernels), input 56×56×128 → output 56×56×256 (conv3_1, conv3_2, conv3_3)
- 3 Conv layers: 512 filters (3×3 kernels), input 28×28×256 → output 28×28×512 (conv4_1, conv4_2, conv4_3)
- 3 Conv layers: 512 filters (3×3 kernels), input 14×14×512 → output 14×14×512 (conv5_1, conv5_2, conv5_3)
- Followed by max pooling layers after each group.

2. **Fully Connected Layers**:

- 1 FC layer: 4096 neurons
- 1 FC layer: 4096 neurons
- 1 FC layer: 1000 neurons (for classification)

Formula for Parameters:

For convolutional layers:

$$Parameters = \left(n_{filters} \times n_{input\ channels} \times kernel\ height \times kernel\ width\right) + n_{filters}\ (for\ biases)$$

For fully connected layers:

$$Parameters = (input\ units \times output\ units) + output\ units$$

Step-by-Step Calculation:

1. **Layer 1: conv1_1**: (64×3×3×3)+64=1792
2. **Layer 2: conv1_2**: (64×64×3×3)+64=36,928
3. **Layer 3: conv2_1**: (128×64×3×3)+128=73,856
4. **Layer 4: conv2_2**: (128×128×3×3)+128=147,584
5. **Layer 5: conv3_1**: (256×128×3×3)+256=295,168
6. **Layer 6: conv3_2**: (256×256×3×3)+256=590,080
7. **Layer 7: conv3_3**: (256×256×3×3)+256=590,080
8. **Layer 8: conv4_1**: (512×256×3×3)+512=1,180,160
9. **Layer 9: conv4_2**: (512×512×3×3)+512=2,359,808
10. **Layer 10: conv4_3**:(512×512×3×3)+512=2,359,808
11. **Layer 11: conv5_1**:(512×512×3×3)+512=2,359,808
12. **Layer 12: conv5_2**:(512×512×3×3)+512=2,359,808
13. **Layer 13: conv5_3**:(512×512×3×3)+512=2,359,808

Fully Connected Layers:

1. **Layer 14: FC1**: (7×7×512)×4096+4096=102,764,544
2. **Layer 15: FC2**: 4096×4096+4096=16,781,
3. **Layer 16: FC3**: 4096×1000+1000=4,097,000

Total Number of Parameters:

Summing all the parameters:

1792+36,928+73,856+147,584+295,168+590,080+590,080+1,180,160+2,359,808+2,359,808+2,359,808+2,359,808+2,359,808+102,764,544+16,781,312+4,097,000=138,357,544

So, VGG-16 has approximately 138.3 million parameters.

CONTINUED PROJECT REPORT: SUBMITTED BY BHAVYA TIWARI

1. PLOT IMAGE AND PIXEL VALUE VS FREQUENCY GRAPH

CODE:

infosys_mri.ipynb

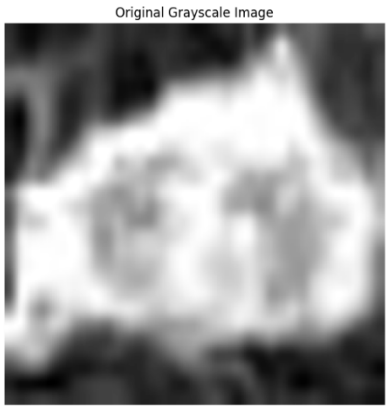
File Edit View Insert Runtime Tools Help All changes saved

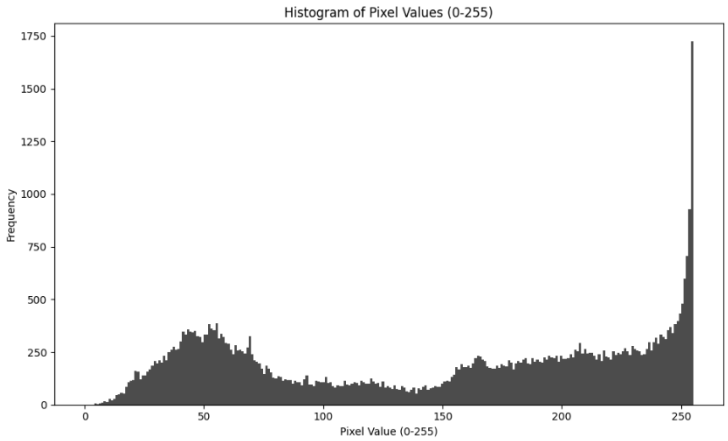
+ Code + Text

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from skimage.feature import hog
from skimage import exposure

# Load the image
image_path = '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaDM-Ma-2127/SUB7/p-049.jpg'
image = Image.open(image_path).convert('L')  # Convert to grayscale

# Resize the image to 224x224
image = image.resize((224, 224))

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

# Plot the original image, HOG image, and histogram of pixel values
plt.figure(figsize=(18, 6))

# Display the original grayscale image
plt.subplot(1, 2, 1)
plt.imshow(image_np, cmap='gray')
plt.title('Original Grayscale Image')
plt.axis('off')


# Plot the histogram of pixel values (0-255)
plt.subplot(1, 2,2)
plt.hist(image_np.ravel(), bins=256, range=(0, 255), color='black', alpha=0.7)
plt.title('Histogram of Pixel Values (0-255)')
plt.xlabel('Pixel Value (0-255)')
plt.ylabel('Frequency')

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

✓ 2s completed at 11:09 PM

OUTPUT:





2. HOG FEATURE EXTRACTION

CODE:

infosys_mri.ipynb

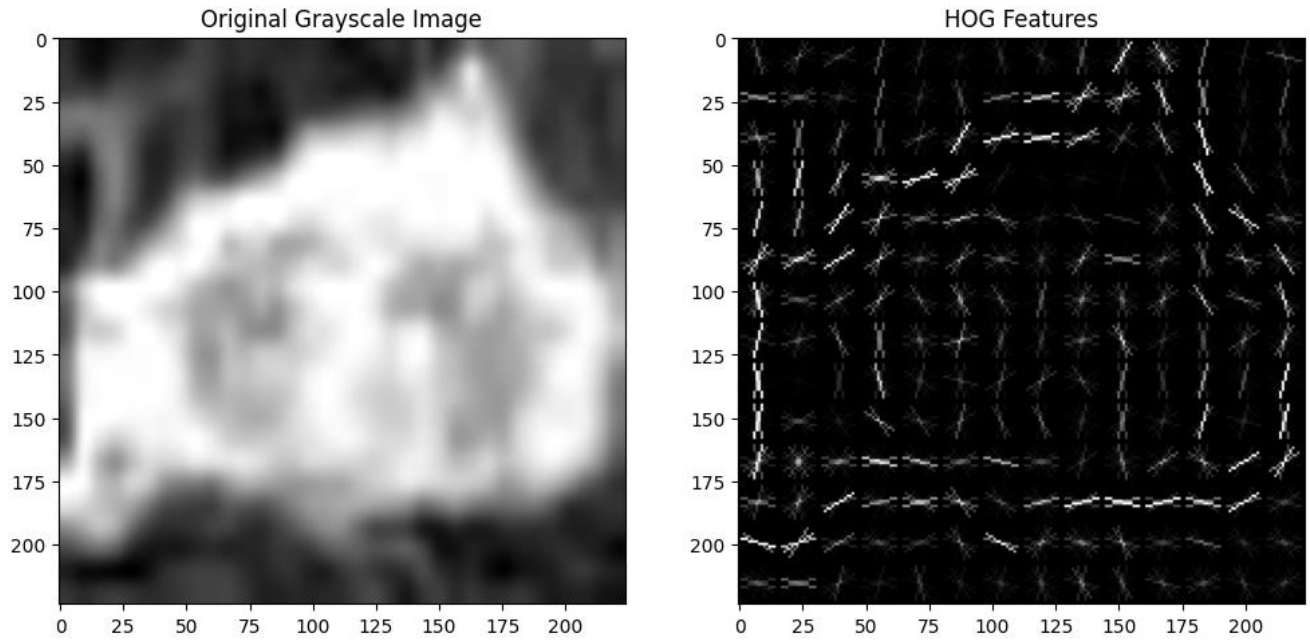
File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from skimage.feature import hog
from skimage import exposure

# Load the image
image_path = '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaDM-Ma-2127/SUB7/p-049.jpg'
image = Image.open(image_path).convert('L')  # Convert to grayscale
image = image.resize((224, 224))
image_np = np.array(image)

# Perform HOG feature extraction
hog_features, hog_image = hog(image_np, orientations=8, pixels_per_cell=(16, 16),
                              cells_per_block=(1, 1), visualize=True, channel_axis=None)

# Rescale HOG image for better visualization
hog_image_rescaled = exposure.rescale_intensity(hog_image, in_range=(0, 10))

# Plot the original and HOG images
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(image_np, cmap='gray')
plt.title('Original Grayscale Image')

plt.subplot(1, 2, 2)
plt.imshow(hog_image_rescaled, cmap='gray')
plt.title('HOG Features')

plt.show()

print("Image Array:")
print(image_np)
print("HOG Features Shape:", hog_features.shape)
```

✓ 2s completed at 11:09 PM

OUTPUT:



3. SOBEL OPERATOR

CODE:

infosys_mri.ipynb

File Edit View Insert Runtime Tools Help

+ Code + Text

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

# Load the image
image_path = '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaDM-Ma-2127/SUB7/p-049.jpg'
image = Image.open(image_path).convert('L')  # Convert to grayscale
image = image.resize((224, 224))
image_np = np.array(image)

# 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 perform convolution using the kernel
def convolve(x, h):
    xh, xw = x.shape
    hh, hw = h.shape
    # Kernel radius
    rh, rw = np.array(h.shape)//2
    # Init output
    output = np.zeros(x.shape)
    for n1 in prange(rh, xh-rh):
        for n2 in prange(rw, xw-rw):
            value = 0
            for k1 in prange(hh):
                for k2 in prange(hw):
                    value += h[k1, k2]*x[n1 + k1 - rh, n2 + k2 - rw]
            output[n1, n2] = value
    return output
```

✓ 2s completed at 11:09 PM

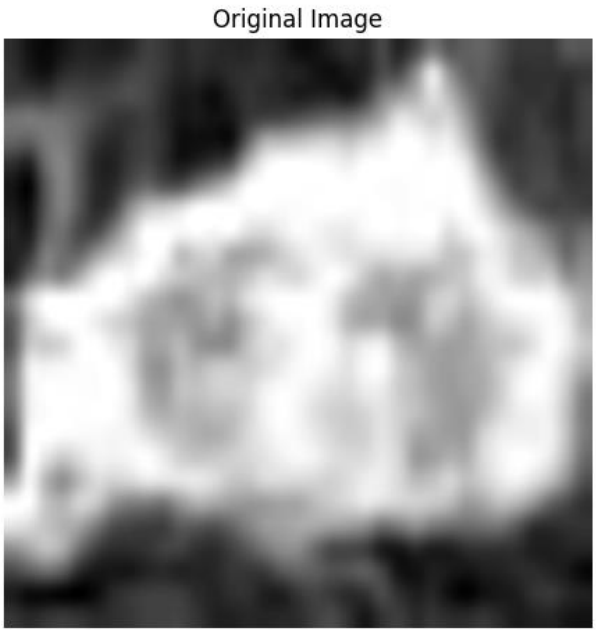
infosys_mri.ipynb

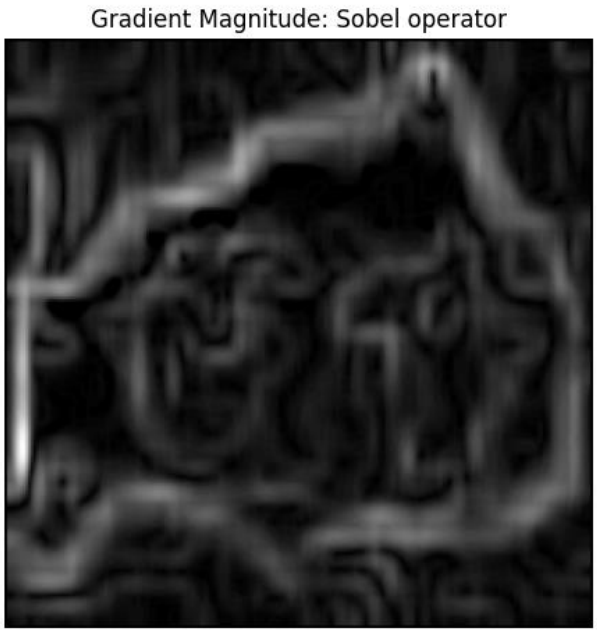
File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
# Apply convolution using Sobel X and Y kernels
gradient_x = convolve(image_np, sobel_x)
gradient_y = convolve(image_np, sobel_y)

# Compute the gradient magnitude
gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)

# Normalize the result for visualization (to range 0-1)
gradient_magnitude = (gradient_magnitude - gradient_magnitude.min()) / (gradient_magnitude.max() - gradient_magnitude.min())

# Display the original and gradient magnitude images
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title('Original Image')
plt.imshow(image_np, cmap='gray')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title('Gradient Magnitude: Sobel operator')
plt.imshow(gradient_magnitude, cmap='gray')
plt.axis('off')

plt.show()
```

OUTPUT:





4. LOCAL BINARY PATTERN

CODE:

infosys_mri.ipynb

File Edit View Insert Runtime Tools Help

+ Code + Text

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

# Load the image (Ensure path correctness)
image_path = '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaDM-Ma-2127/SUB7/p-049.jpg'
image = Image.open(image_path).convert('L')  # Convert to grayscale
image = image.resize((224, 224))
image_np = np.array(image)

def getLBPimage(gray_image):
    '''
    == Input ==
    gray_image  : grayscale image of shape (height, width)

    == Output ==
    imgLBP : LBP converted image of the same shape
    '''
    # Initialize LBP image with zeros
    imgLBP = np.zeros_like(gray_image)
    neighboor = 3  # Define a 3x3 neighborhood

    # Iterate over each pixel (excluding borders)
    for ih in range(1, gray_image.shape[0] - 1):
        for iw in range(1, gray_image.shape[1] - 1):
            # Step 1: Get the 3x3 neighborhood around the pixel
            img = gray_image[ih-1:ih+2, iw-1:iw+2]  # 3x3 window
            center = img[1, 1]  # The center pixel
            # Step 2: Create a binary pattern by thresholding
            img01 = (img >= center).astype(int)
            # Step 3: Flatten the matrix into a vector (excluding the center)
            img01_vector = np.delete(img01.flatten(), 4)
            # Step 4: Convert the binary pattern to a decimal number
            num = np.dot(img01_vector, 2 ** np.arange(8))
            # Step 5: Assign this number to the LBP image
            imgLBP[ih, iw] = num
```

✓ 2s completed at 11:09 PM

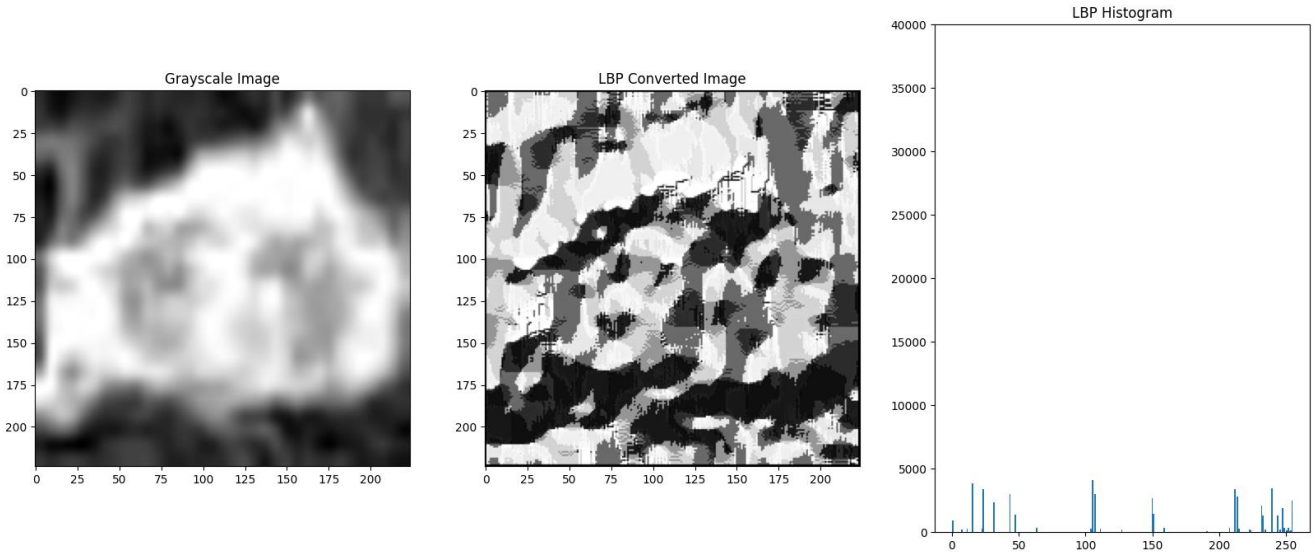
infosys_mri.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

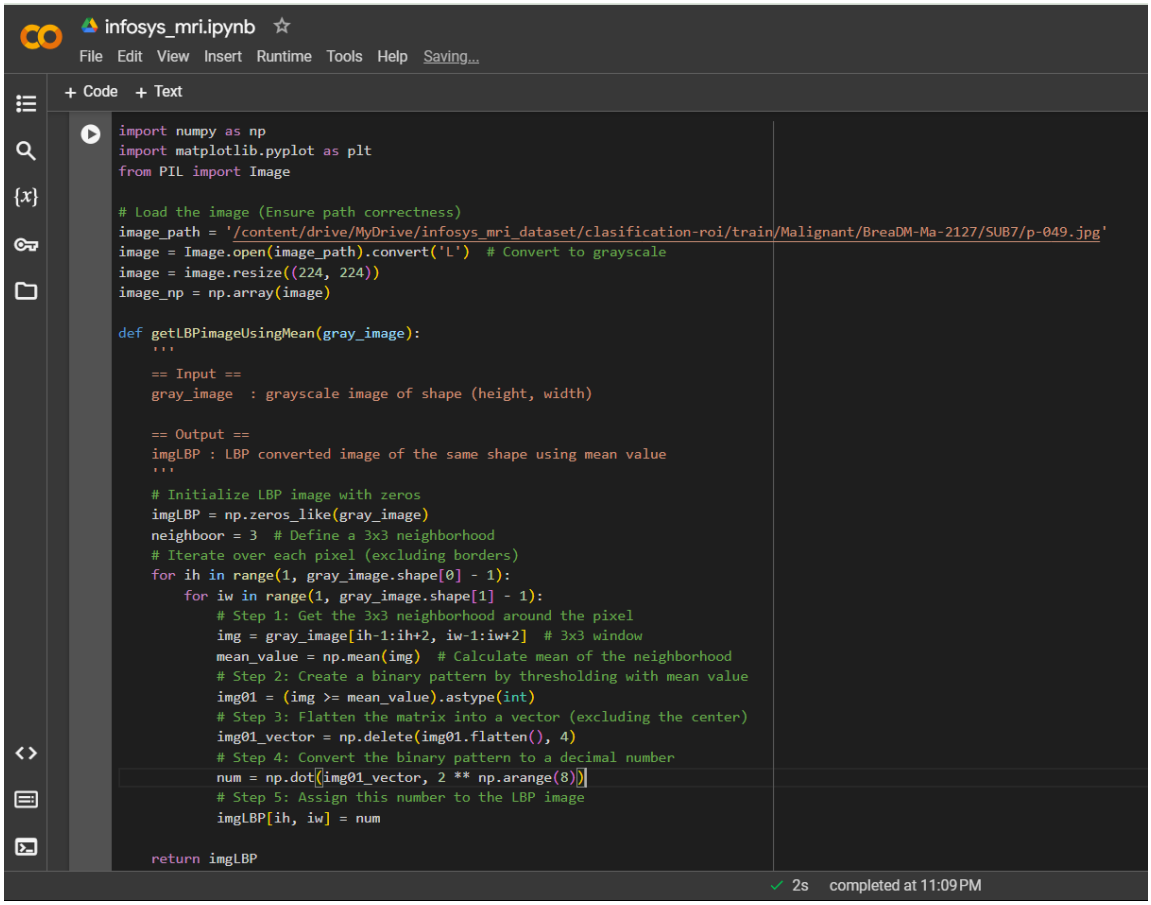
```
    return imgLBP

# Apply LBP to the single image
imgLBP = getLBPimage(image_np)

# Flatten the LBP image for histogram
vecimgLBP = imgLBP.flatten()

# Plot the original image, LBP image, and histogram
fig = plt.figure(figsize=(20, 8))

# Plot original grayscale image
ax = fig.add_subplot(1, 3, 1)
ax.imshow(image_np, cmap="gray")
ax.set_title("Grayscale Image")

# Plot LBP converted image
ax = fig.add_subplot(1, 3, 2)
ax.imshow(imgLBP, cmap="gray")
ax.set_title("LBP Converted Image")

# Plot LBP histogram
ax = fig.add_subplot(1, 3, 3)
freq, lbp, _ = ax.hist(vecimgLBP, bins=256)
ax.set_ylim(0, 40000)
lbp = lbp[:-1]

# Print LBP values with high frequencies
largeTF = freq > 5000
for x, fr in zip(lbp[largeTF], freq[largeTF]):
    ax.text(x, fr, "{:6.0f}".format(x), color="magenta")
ax.set_title("LBP Histogram")

plt.show()
```

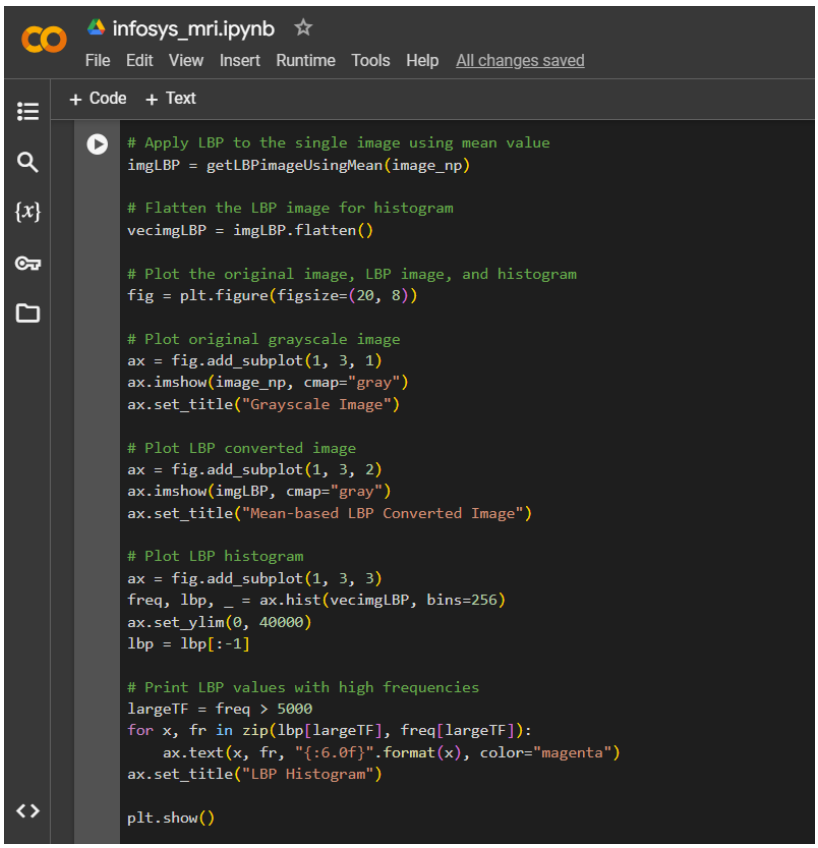
OUTPUT:



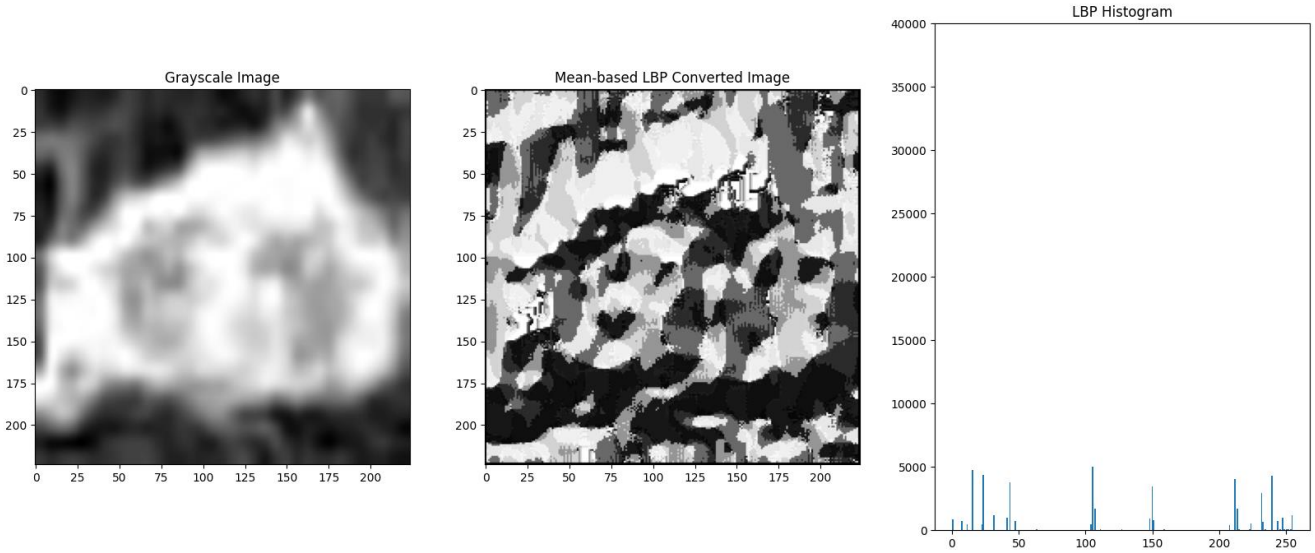
5. MEAN-BASED LBP:

CODE:





OUTPUT:



6. MEDIAN-BASED LBP CONVERTED IMAGE

CODE:

infosys_mri.ipynb

File Edit View Insert Runtime Tools Help Saving...

+ Code + Text

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

# Load the image (Ensure path correctness)
image_path = '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaDM-Ma-2127/SUB7/p-049.jpg'
image = Image.open(image_path).convert('L')
image = image.resize((224, 224))
image_np = np.array(image)

def getLBPimageUsingMedian(gray_image):
    '''
    == Input ==
    gray_image  : grayscale image of shape (height, width)

    == Output ==
    imgLBP : LBP converted image of the same shape using median value
    '''
    # Initialize LBP image with zeros
    imgLBP = np.zeros_like(gray_image)
    neighboor = 3  # Define a 3x3 neighborhood
    # Iterate over each pixel (excluding borders)
    for ih in range(1, gray_image.shape[0] - 1):
        for iw in range(1, gray_image.shape[1] - 1):
            # Step 1: Get the 3x3 neighborhood around the pixel
            img = gray_image[ih-1:ih+2, iw-1:iw+2]  # 3x3 window
            median_value = np.median  (img)  # Calculate median of the neighborhood
            # Step 2: Create a binary pattern by thresholding with median value
            img01 = (img >= median_value).astype(int)
            # Step 3: Flatten the matrix into a vector (excluding the center)
            img01_vector = np.delete(img01.flatten(), 4)
            # Step 4: Convert the binary pattern to a decimal number
            num = np.dot(img01_vector, 2 ** np.arange(8))
            # Step 5: Assign this number to the LBP image
            imgLBP[ih, iw] = num

    return imgLBP
```

✓ 2s completed at 11:09 PM

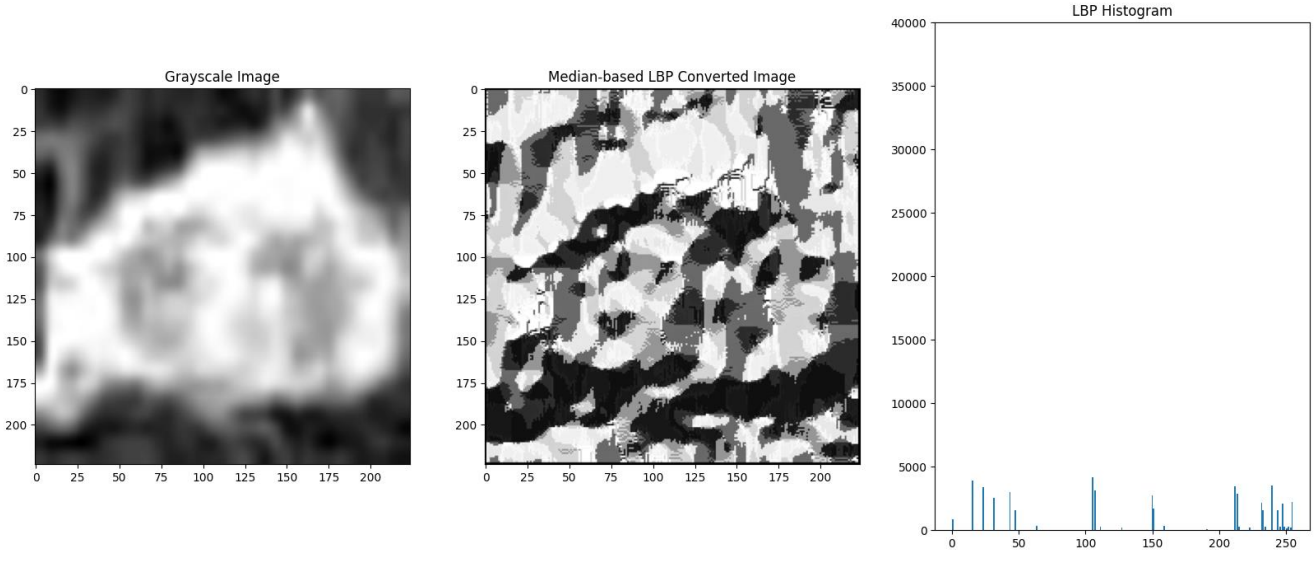
infosys_mri.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
# Apply LBP to the single image using median value
imgLBP = getLBPimageUsingMedian(image_np)

# Flatten the LBP image for histogram
vecimgLBP = imgLBP.flatten()

# Plot the original image, LBP image, and histogram
fig = plt.figure(figsize=(20, 8))

# Plot original grayscale image
ax = fig.add_subplot(1, 3, 1)
ax.imshow(image_np, cmap="gray")
ax.set_title("Grayscale Image")

# Plot LBP converted image
ax = fig.add_subplot(1, 3, 2)
ax.imshow(imgLBP, cmap="gray")
ax.set_title("Median-based LBP Converted Image")

# Plot LBP histogram
ax = fig.add_subplot(1, 3, 3)
freq, lbp, _ = ax.hist(vecimgLBP, bins=256)
ax.set_ylim(0, 40000)
lbp = lbp[:-1]

# Print LBP values with high frequencies
largeTF = freq > 5000
for x, fr in zip(lbp[largeTF], freq[largeTF]):
    ax.text(x, fr, "{:6.0f}".format(x), color="magenta")
ax.set_title("LBP Histogram")

plt.show()
```

OUTPUT:



7. VARIANCE-BASED LBP:

CODE:

infosys_mri.ipynb

File Edit View Insert Runtime Tools Help Saving...

+ Code + Text

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

# Load the image (Ensure path correctness)
image_path = '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaDM-Ma-2127/SUB7/p-049.jpg'
image = Image.open(image_path).convert('L')
image = image.resize((224, 224))
image_np = np.array(image)

def getLBPimageUsingVariance(gray_image):
    '''
    == Input ==
    gray_image  : grayscale image of shape (height, width)

    == Output ==
    imgLBP : LBP converted image of the same shape using variance
    '''
    # Initialize LBP image with zeros
    imgLBP = np.zeros_like(gray_image)
    neighboor = 3  # Define a 3x3 neighborhood
    # Iterate over each pixel (excluding borders)
    for ih in range(1, gray_image.shape[0] - 1):
        for iw in range(1, gray_image.shape[1] - 1):
            # Step 1: Get the 3x3 neighborhood around the pixel
            img = gray_image[ih-1:ih+2, iw-1:iw+2]  # 3x3 window
            variance_value = np.var(img)  # Calculate variance of the neighborhood
            # Step 2: Create a binary pattern by thresholding with the variance
            img01 = (img >= variance_value).astype(int)
            # Step 3: Flatten the matrix into a vector (excluding the center)
            img01_vector = np.delete(img01.flatten(), 4)
            # Step 4: Convert the binary pattern to a decimal number
            num = np.dot(img01_vector, 2 ** np.arange(8))
            # Step 5: Assign this number to the LBP image
            imgLBP[ih, iw] = num

    return imgLBP
```

✓ 2s completed at 11:09 PM

infosys_mri.ipynb

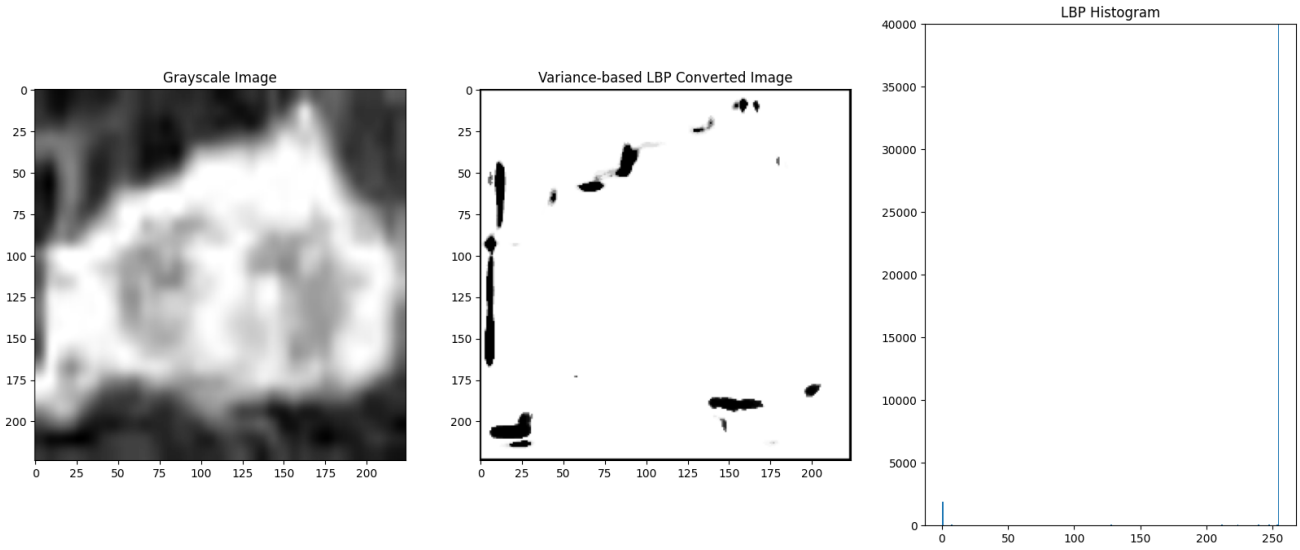
File Edit View Insert Runtime Tools Help

+ Code + Text

```
# Apply LBP to the single image using variance
imgLBP = getLBPimageUsingVariance(image_np)
# Flatten the LBP image for histogram
vecimgLBP = imgLBP.flatten()
# Plot the original image, LBP image, and histogram
fig = plt.figure(figsize=(20, 8))
# Plot original grayscale image
ax = fig.add_subplot(1, 3, 1)
ax.imshow(image_np, cmap="gray")
ax.set_title("Grayscale Image")
# Plot LBP converted image
ax = fig.add_subplot(1, 3, 2)
ax.imshow(imgLBP, cmap="gray")
ax.set_title("Variance-based LBP Converted Image")
# Plot LBP histogram
ax = fig.add_subplot(1, 3, 3)
freq, lbp, _ = ax.hist(vecimgLBP, bins=256)
ax.set_ylim(0, 40000)
lbp = lbp[:-1]
# Print LBP values with high frequencies
largeTF = freq > 5000
for x, fr in zip(lbp[largeTF], freq[largeTF]):
    ax.text(x, fr, "{:6.0f}".format(x), color="magenta")
ax.set_title("LBP Histogram")
plt.show()
```

OUTPUT:

254



8. MVM-BASED LBP:

CODE:

infosys_mri.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
# Load the image (Ensure path correctness)
image_path = '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaDM-Ma-2127/SUB7/p-049.jpg'
image = Image.open(image_path).convert('L')
image = image.resize((224, 224))
image_np = np.array(image)

def getLBPimageUsingMVM(gray_image):
    '''
    == Input ==
    gray_image  : grayscale image of shape (height, width)

    == Output ==
    imgLBP : LBP converted image of the same shape using Mean-Variance-Median (MVM) metric
    '''
    # Initialize LBP image with zeros
    imgLBP = np.zeros_like(gray_image)

    # Iterate over each pixel (excluding borders)
    for ih in range(1, gray_image.shape[0] - 1):
        for iw in range(1, gray_image.shape[1] - 1):
            # Step 1: Get the 3x3 neighborhood around the pixel
            img = gray_image[ih-1:ih+2, iw-1:iw+2]  # 3x3 window
            # Step 2: Calculate Mean, Variance, and Median of the neighborhood
            mean_value = np.mean(img)
            variance_value = np.var(img)
            median_value = np.median(img)
            # Step 3: Calculate MVM threshold
            mvm_threshold = (mean_value + np.sqrt(variance_value) + median_value) / 3
            # Step 4: Create a binary pattern by thresholding with MVM threshold
            img01 = (img >= mvm_threshold).astype(int)
            # Step 5: Flatten the matrix into a vector (excluding the center)
            img01_vector = np.delete(img01.flatten(), 4)
            # Step 6: Convert the binary pattern to a decimal number
            num = np.dot(img01_vector, 2 ** np.arange(8))
            # Step 7: Assign this number to the LBP image
            imgLBP[ih, iw] = num
    return imgLBP
```

✓ 2s completed at 11:09 PM

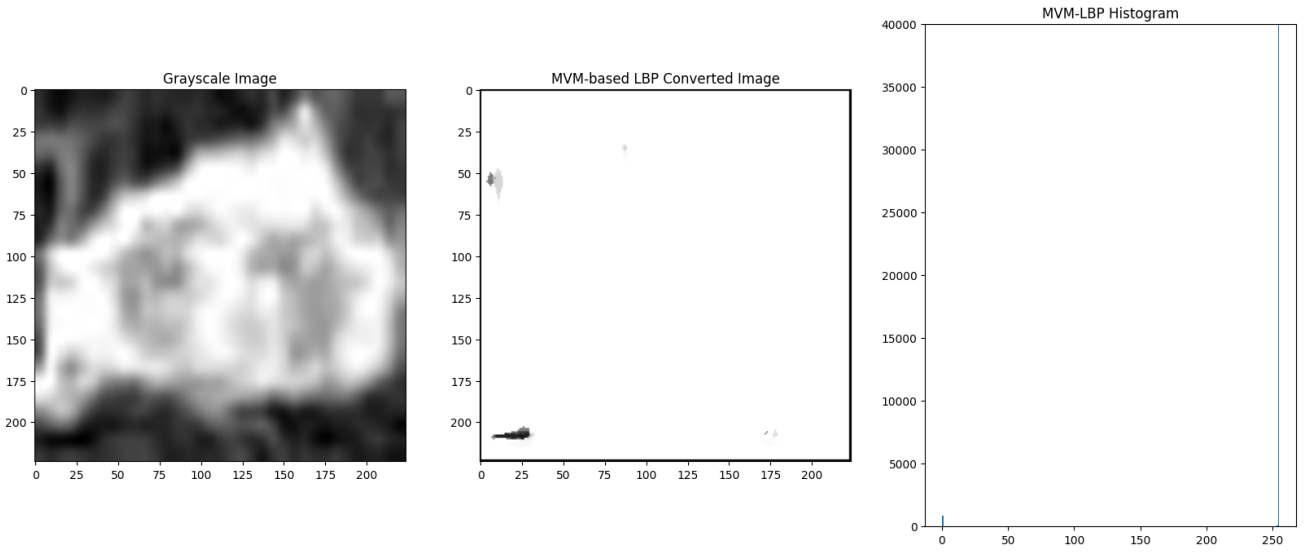
infosys_mri.ipynb

File Edit View Insert Runtime Tools Help Saving...

+ Code + Text

```
# Apply MVM-LBP to the single image
imgLBP = getLBPimageUsingMVM(image_np)
# Flatten the MVM-LBP image for histogram
vecimgLBP = imgLBP.flatten()
# Plot the original image, MVM-LBP image, and histogram
fig = plt.figure(figsize=(20, 8))
# Plot original grayscale image
ax = fig.add_subplot(1, 3, 1)
ax.imshow(image_np, cmap="gray")
ax.set_title("Grayscale Image")
# Plot MVM-based LBP converted image
ax = fig.add_subplot(1, 3, 2)
ax.imshow(imgLBP, cmap="gray")
ax.set_title("MVM-based LBP Converted Image")
# Plot MVM-LBP histogram
ax = fig.add_subplot(1, 3, 3)
freq, lbp, _ = ax.hist(vecimgLBP, bins=256)
ax.set_ylim(0, 40000)
lbp = lbp[:-1]

# Print MVM-LBP values with high frequencies
largeTF = freq > 5000
for x, fr in zip(lbp[largeTF], freq[largeTF]):
    ax.text(x, fr, "{:6.0f}".format(x), color="magenta")
ax.set_title("MVM-LBP Histogram")

plt.show()
```

OUTPUT:



9. MVM BASED LBP FOR 5 IMAGES:

CODE:

infosys_mri.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

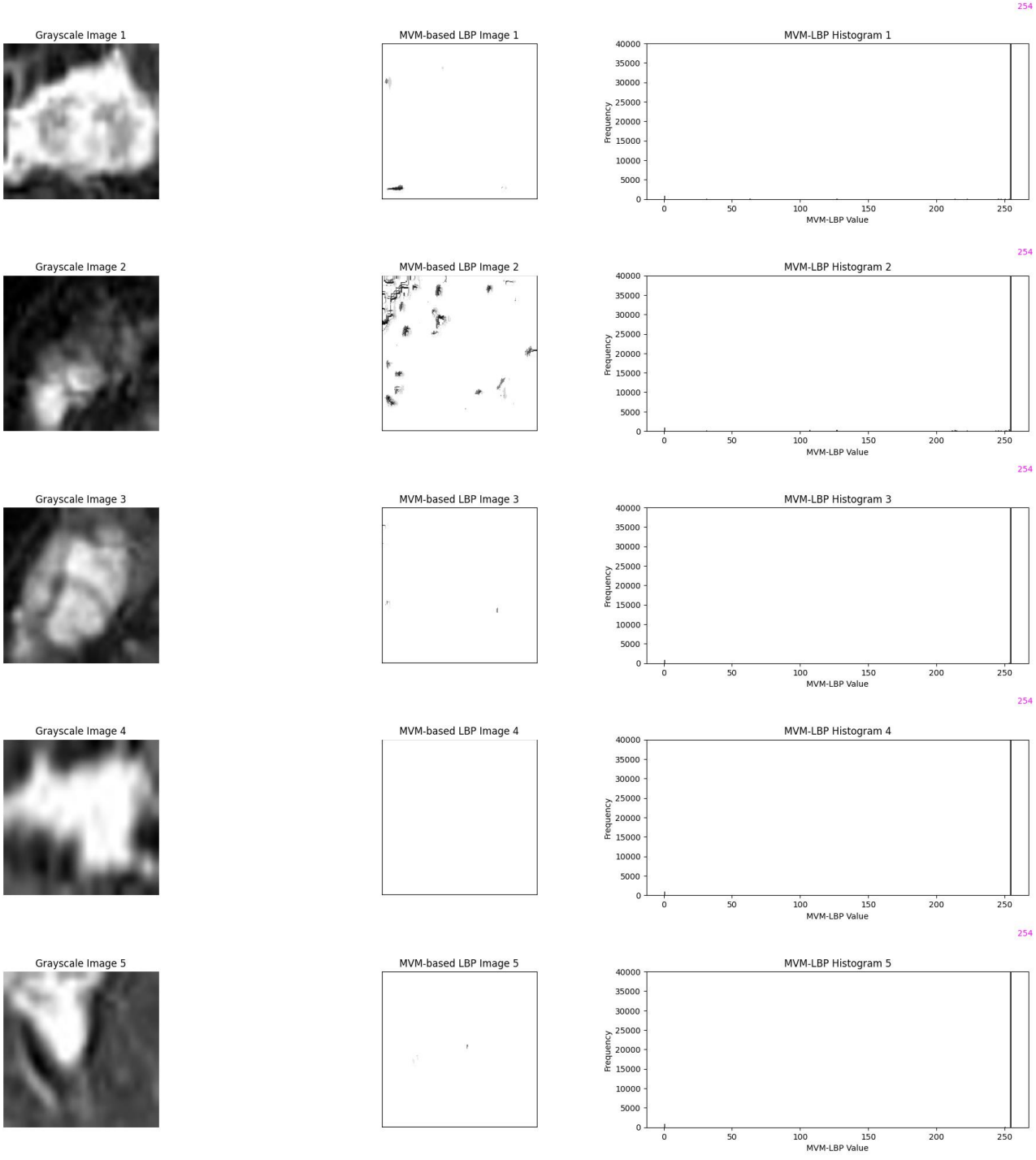
```
[14] def getLBPimageUsingMVM(gray_image):
         '''
         == Input ==
         gray_image  : grayscale image of shape (height, width)

         == Output ==
         imgLBP : LBP converted image of the same shape using Mean-Variance-Median (MVM) metric
         '''
         # Initialize LBP image with zeros
         imgLBP = np.zeros_like(gray_image)

         # Iterate over each pixel (excluding borders)
         for ih in range(1, gray_image.shape[0] - 1):
             for iw in range(1, gray_image.shape[1] - 1):
                 # Step 1: Get the 3x3 neighborhood around the pixel
                 img = gray_image[ih-1:ih+2, iw-1:iw+2]  # 3x3 window
                 # Step 2: Calculate Mean, Variance, and Median of the neighborhood
                 mean_value = np.mean(img)
                 variance_value = np.var(img)
                 median_value = np.median(img)
                 # Step 3: Calculate MVM threshold
                 mvm_threshold = (mean_value + np.sqrt(variance_value) + median_value) / 3
                 # Step 4: Create a binary pattern by thresholding with MVM threshold
                 img01 = (img >= mvm_threshold).astype(int)
                 # Step 5: Flatten the matrix into a vector (excluding the center)
                 img01_vector = np.delete(img01.flatten(), 4)
                 # Step 6: Convert the binary pattern to a decimal number
                 num = np.dot(img01_vector, 2 ** np.arange(8))
                 # Step 7: Assign this number to the LBP image
                 imgLBP[ih, iw] = num
         return imgLBP
```

+ Code + Text

```
# List of image paths
image_paths = [
    '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaDM-Ma-2127/SUB7/p-049.jpg',
    '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Benign/BreaDM-Be-1805/SUB1/p-025.jpg',
    '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Benign/BreaDM-Be-1805/VIBRANT+C1/p-029.jpg',
    '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaDM-Ma-2117/SUB5/p-085.jpg',
    '/content/drive/MyDrive/infosys_mri_dataset/clasification-roi/train/Malignant/BreaDM-Ma-2128/SUB8/p-052.jpg' ]

# Create a figure for plotting
fig, axs = plt.subplots(len(image_paths), 3, figsize=(20, 4 * len(image_paths)))

# Process each image
for idx, image_path in enumerate(image_paths):
    # Load and preprocess the image
    image = Image.open(image_path).convert('L')
    image = image.resize((224, 224))
    image_np = np.array(image)

    # Apply MVM-LBP to the single image
    imgLBP = getLBPimageUsingMVM(image_np)
    # Flatten the MVM-LBP image for histogram
    vecimgLBP = imgLBP.flatten()

    # Plot original grayscale image
    axs[idx, 0].imshow(image_np, cmap="gray")
    axs[idx, 0].set_title(f"Grayscale Image {idx + 1}")
    axs[idx, 0].axis('off')

    # Plot MVM-based LBP converted image
    axs[idx, 1].imshow(imgLBP, cmap="gray")
    axs[idx, 1].set_title(f"MVM-based LBP Image {idx + 1}")
    axs[idx, 1].axis('off')

    # Plot MVM-LBP histogram
    freq, lbp, _ = axs[idx, 2].hist(vecimgLBP, bins=256, color='black', alpha=0.7)
    axs[idx, 2].set_ylim(0, 40000)
    lbp = lbp[:-1]
```

```
    # Print MVM-LBP values with high frequencies
    largeTF = freq > 5000
    for x, fr in zip(lbp[largeTF], freq[largeTF]):
        axs[idx, 2].text(x, fr, "{:6.0f}".format(x), color="magenta")

    axs[idx, 2].set_title(f"MVM-LBP Histogram {idx + 1}")
    axs[idx, 2].set_xlabel("MVM-LBP Value")
    axs[idx, 2].set_ylabel("Frequency")

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

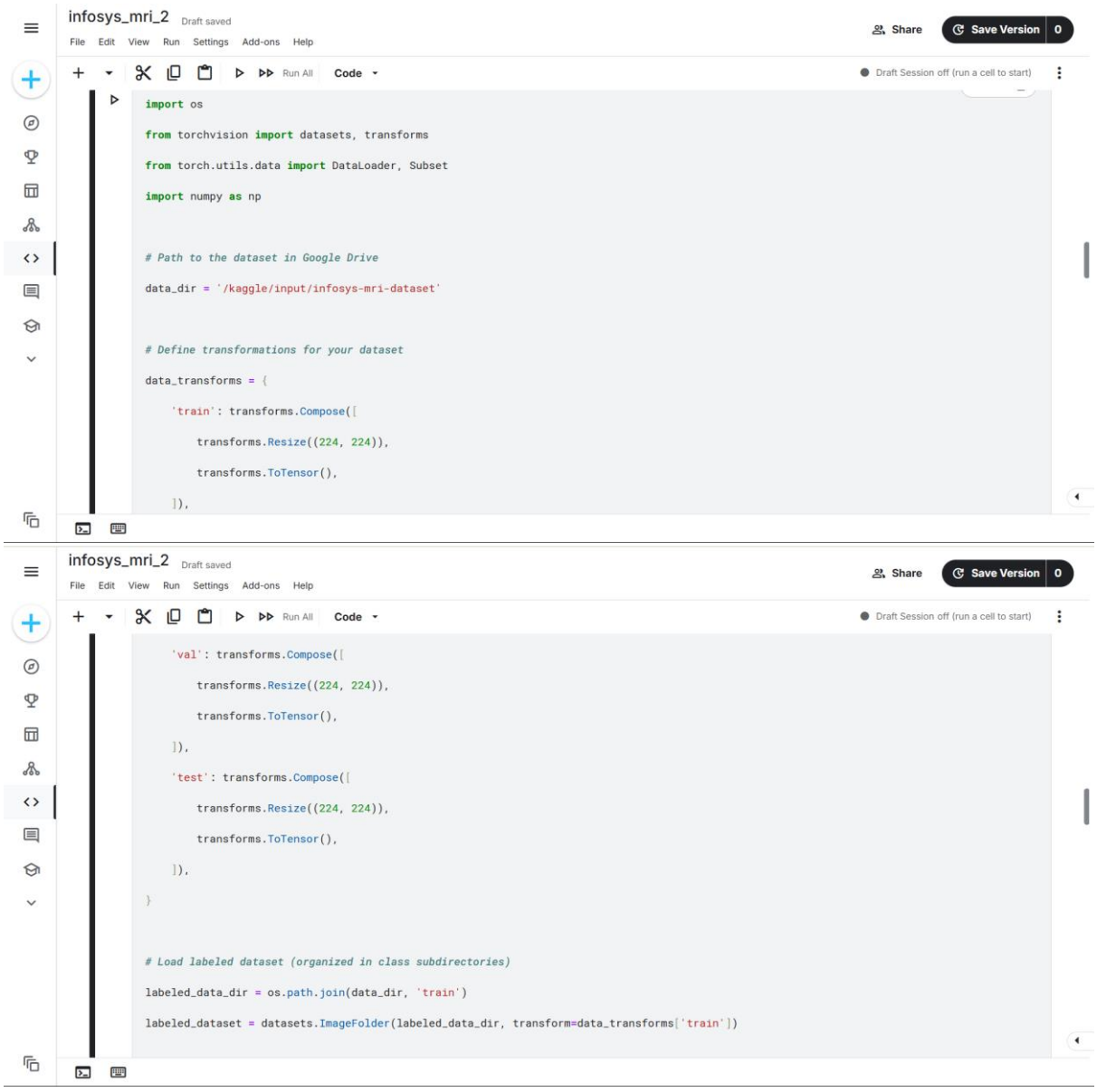
OUTPUT:



POST MVM-LBP CONTINUED PROJECT REPORT: SUBMITTED BY BHAVYA TIWARI

1. **Explored GCLM however outputs were not favourable**

2. **Check for unlabelled data, create data loaders**



infosys_mri_2 Draft saved

File Edit View Run Settings Add-ons Help

Share Save Version 0

Run All Code Draft Session off (run a cell to start)

```
# Extract indices for labeled data
labeled_indices = np.arange(len(labeled_dataset))
X_labeled = Subset(labeled_dataset, labeled_indices)  # Features (X) for labeled
y_labeled = [labeled_dataset.targets[i] for i in labeled_indices]  # Labels (y) for labeled


# Load unlabeled dataset if it's in a separate folder (no class subfolders)
unlabeled_data_dir = os.path.join(data_dir, 'unlabeled')
if os.path.exists(unlabeled_data_dir):
    unlabeled_dataset = datasets.ImageFolder(unlabeled_data_dir, transform=data_transforms['train'])
    # All images in this dataset are treated as unlabeled
    X_unlabeled = DataLoader(unlabeled_dataset, batch_size=32, shuffle=True)
else:
    print("Unlabeled dataset directory not found.")
```

infosys_mri_2 Draft saved

File Edit View Run Settings Add-ons Help

Share Save Version 0

Run All Code Draft Session off (run a cell to start)

```
# Create data loaders for labeled and unlabeled
train_loader = DataLoader(X_labeled, batch_size=32, shuffle=True)
val_loader = DataLoader(datasets.ImageFolder(os.path.join(data_dir, 'val'), transform=data_transforms['val']),
                        batch_size=32, shuffle=True)
test_loader = DataLoader(datasets.ImageFolder(os.path.join(data_dir, 'test'), transform=data_transforms['test']),
                         batch_size=32, shuffle=False)


# Store all loaders in a dictionary for easier access
dataloaders = {
    'train': train_loader,
    'val': val_loader,
    'test': test_loader,
    'unlabeled': X_unlabeled if 'X_unlabeled' in locals() else None  # Check if unlabeled data exists
}
```

infosys_mri_2 Draft saved

File Edit View Run Settings Add-ons Help

Share Save Version 0

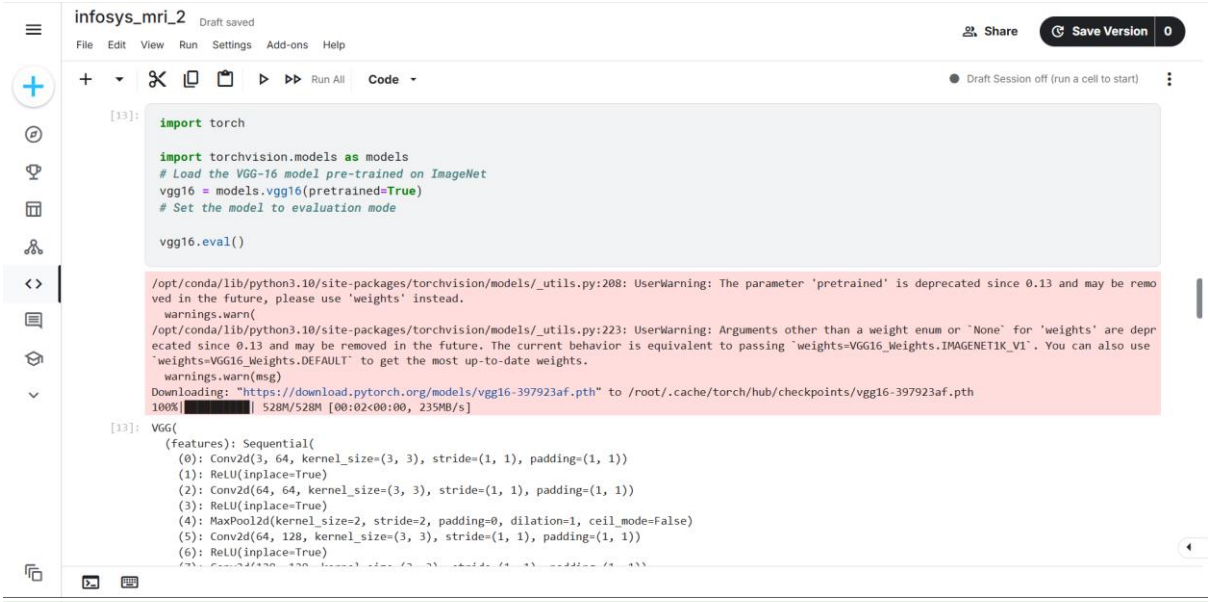
Run All Code Draft Session off (run a cell to start)

```
print("Labeled and unlabeled data loaders created successfully.")
```

```
Unlabeled dataset directory not found.
Labeled and unlabeled data loaders created successfully.
```

+ Code + Markdown

3. Load VGG-16, RESNET-18,RESNET-50



infosys_mri_2 Draft saved

File Edit View Run Settings Add-ons Help

Share

Run All Code Draft Session off (run a ce

```
class Resnet18(nn.Module):
    def __init__(self, num_classes=2):
        super(Resnet18, self).__init__()
        model_resnet18 = models.resnet18(pretrained=True)
        self.conv1 = model_resnet18.conv1  # convolutional function
        self.bn1 = model_resnet18.bn1  # batch normalization
        self.relu = model_resnet18.relu  # relu is your activation function.
        self.maxpool = model_resnet18.maxpool  # maxpool is basically taking the biggest value per

        # sub_matrix
        self.layer1 = model_resnet18.layer1
        self.layer2 = model_resnet18.layer2
        self.layer3 = model_resnet18.layer3
        self.layer4 = model_resnet18.layer4  # these layers are used for deepening the layers in the architecture which will increase

        self.avgpool = model_resnet18.avgpool
        self.features = model_resnet18.fc.in_features
        self.fc = nn.Linear(self.features, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

infosys_mri_2 Draft saved

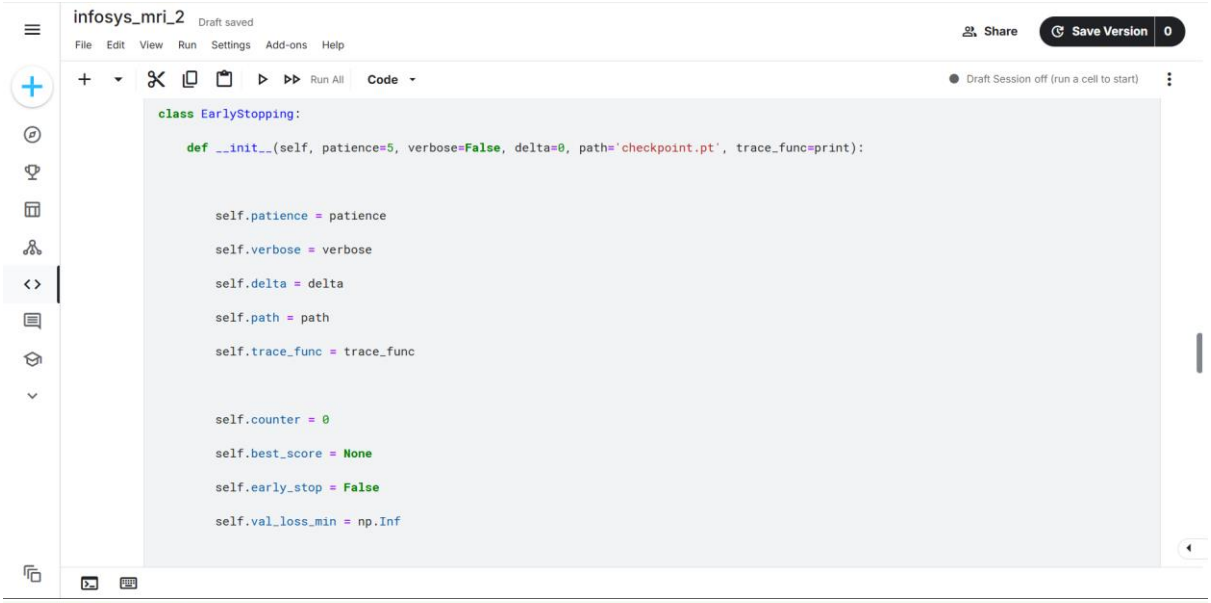
File Edit View Run Settings Add-ons Help

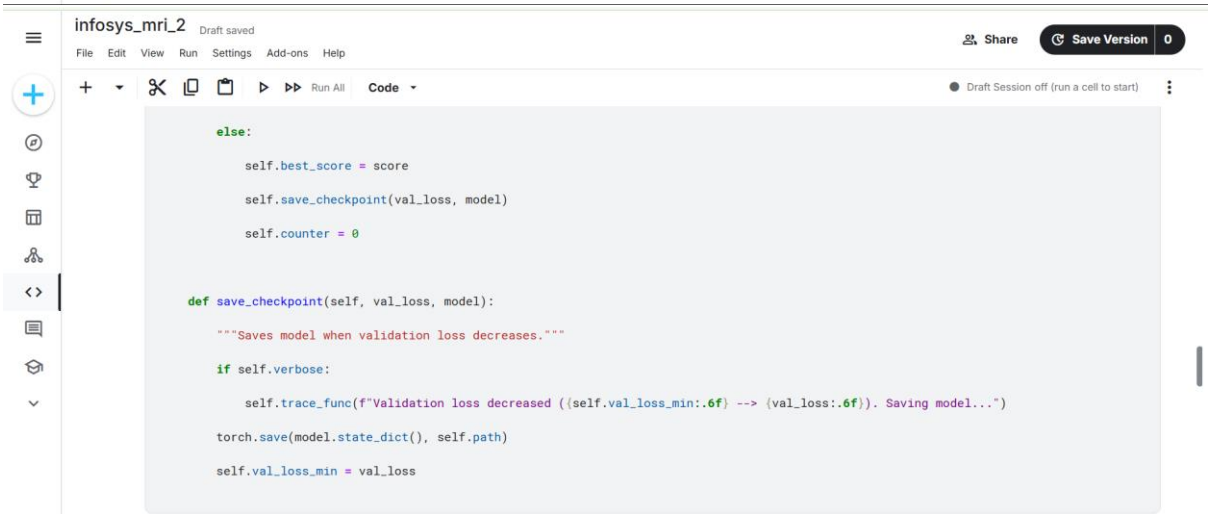
Share

Run All Code Draft Session off (run a ce

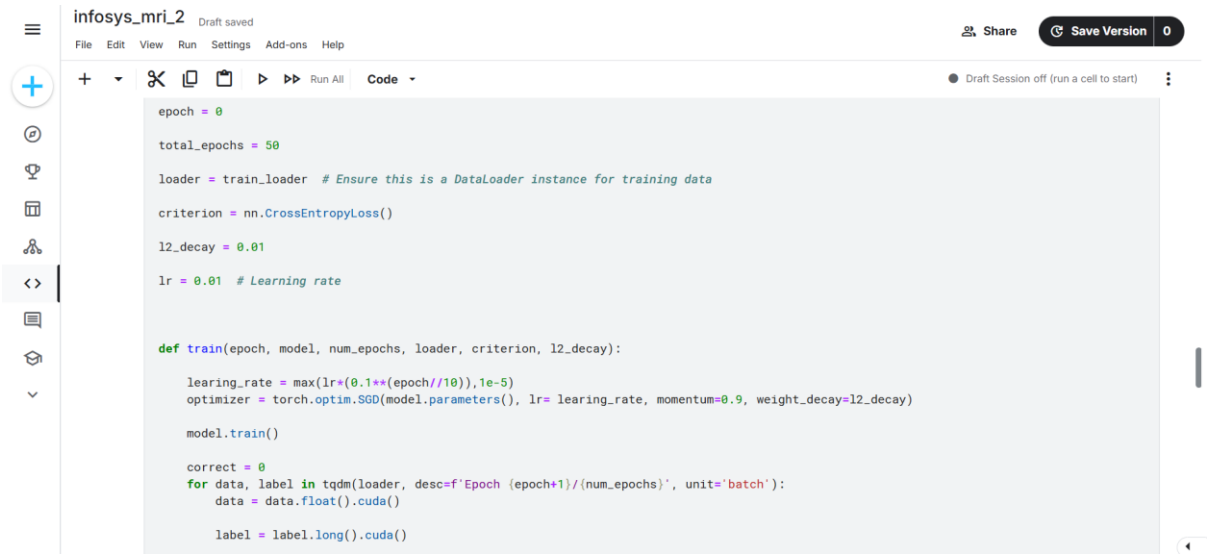
```
class Resnet50(nn.Module):
    def __init__(self, num_classes=2):
        super(Resnet50, self).__init__()
        model_resnet50 = models.resnet50(pretrained=True)
        self.conv1 = model_resnet50.conv1
        self.bn1 = model_resnet50.bn1
        self.relu = model_resnet50.relu
        self.maxpool = model_resnet50.maxpool
        self.layer1 = model_resnet50.layer1
        self.layer2 = model_resnet50.layer2
        self.layer3 = model_resnet50.layer3
        self.layer4 = model_resnet50.layer4
        self.avgpool = model_resnet50.avgpool
        self.features = model_resnet50.fc.in_features
        self.fc = nn.Linear(self.features, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

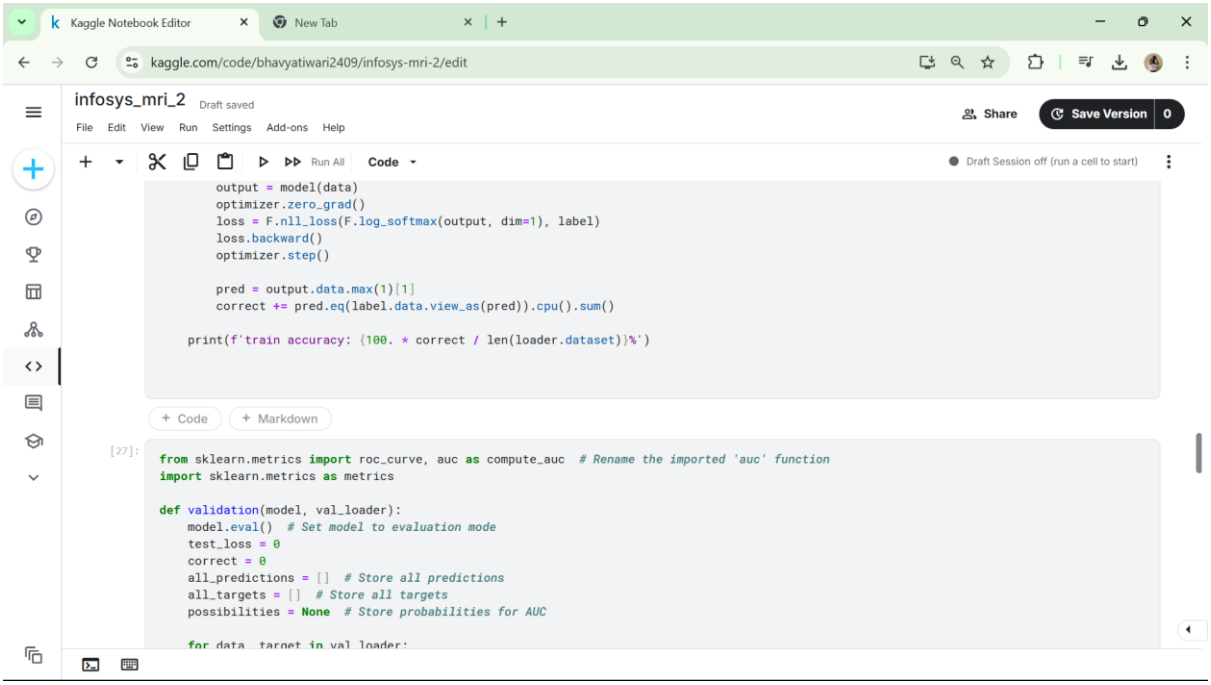
4. Define early stopping

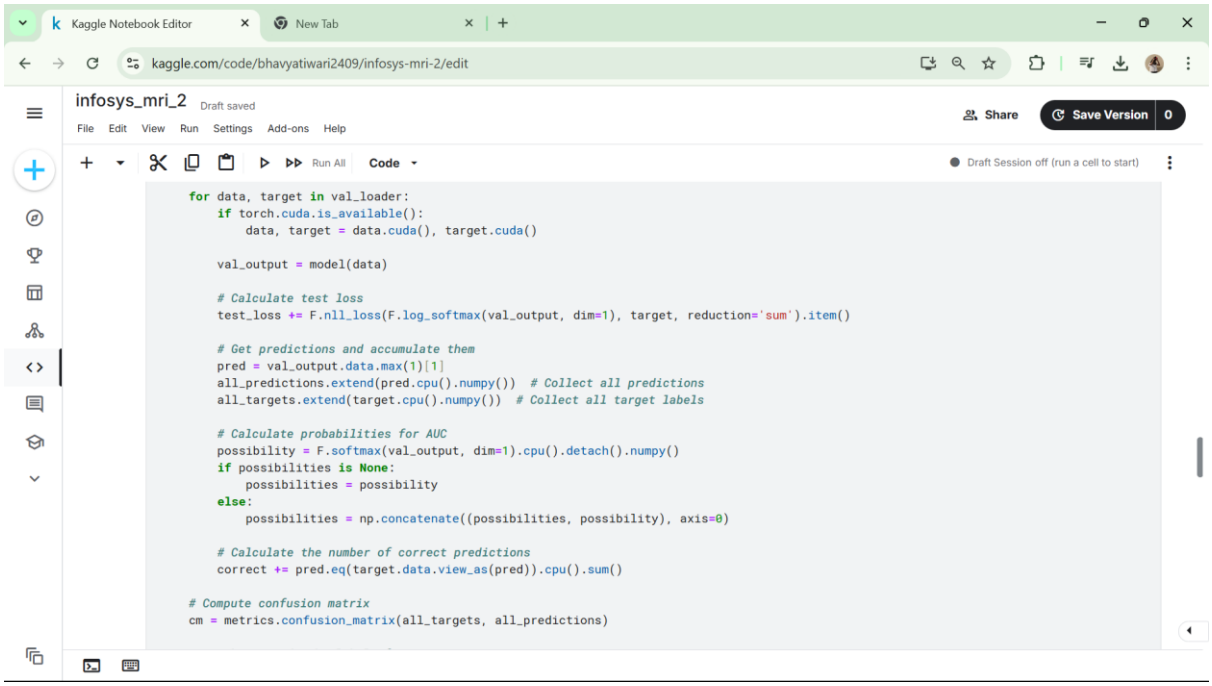


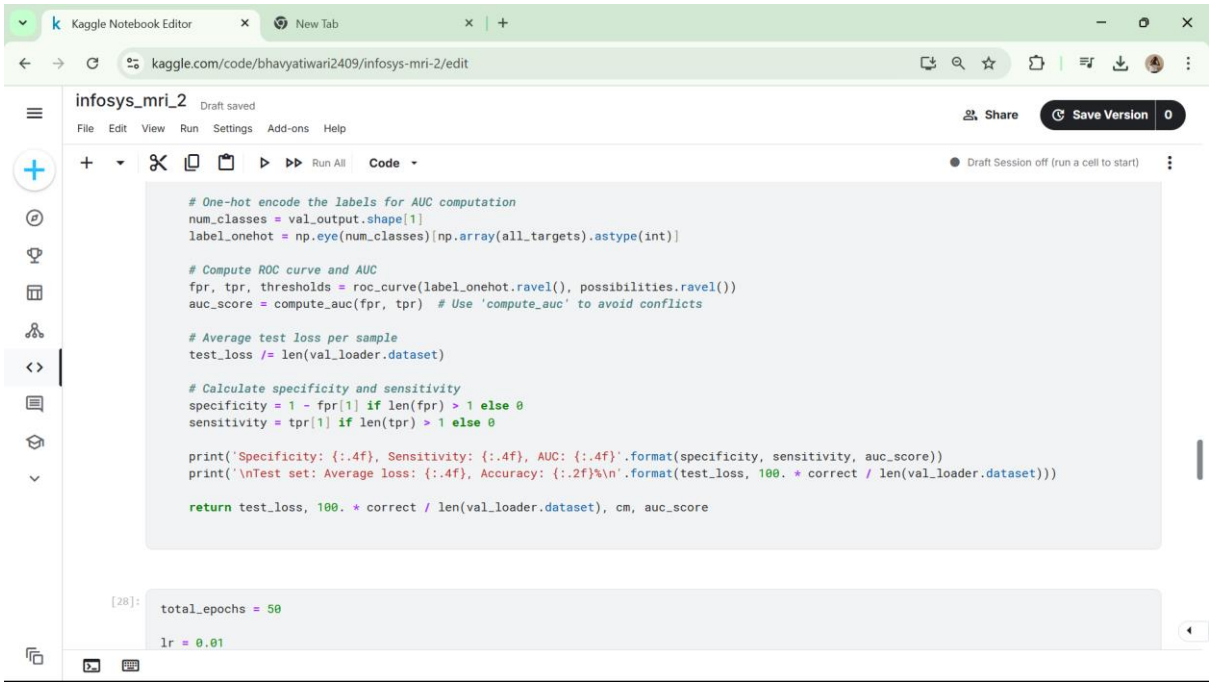


5. Define Train, Validation, Test Functions

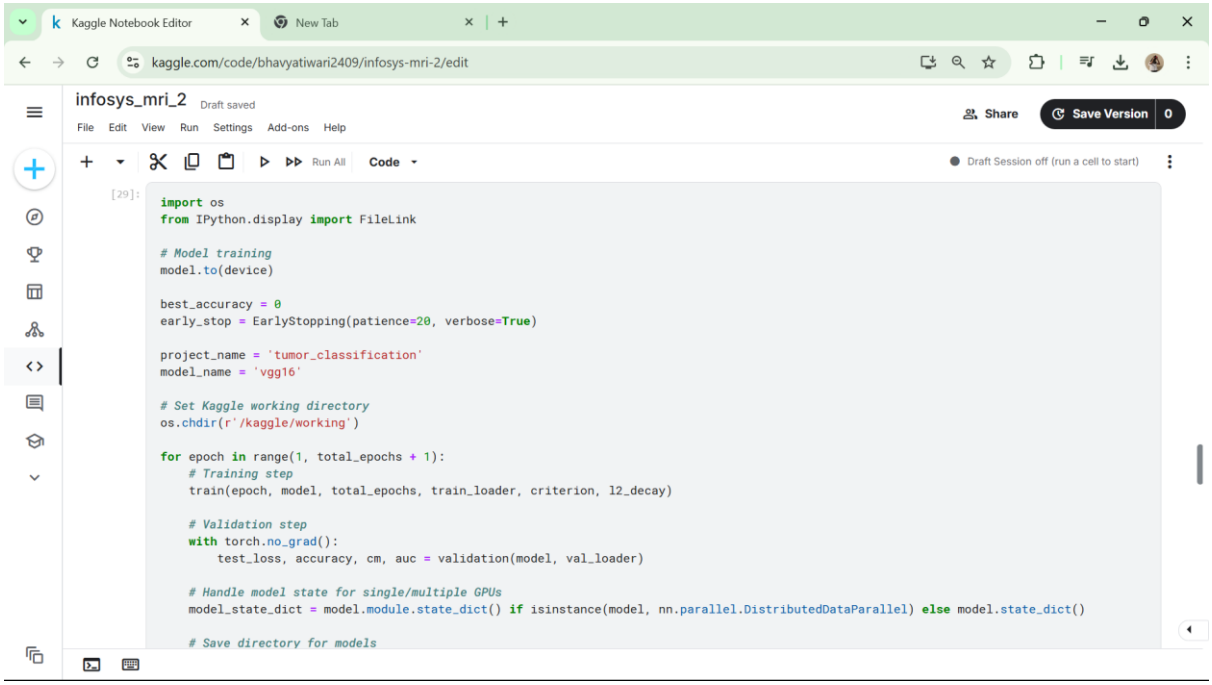


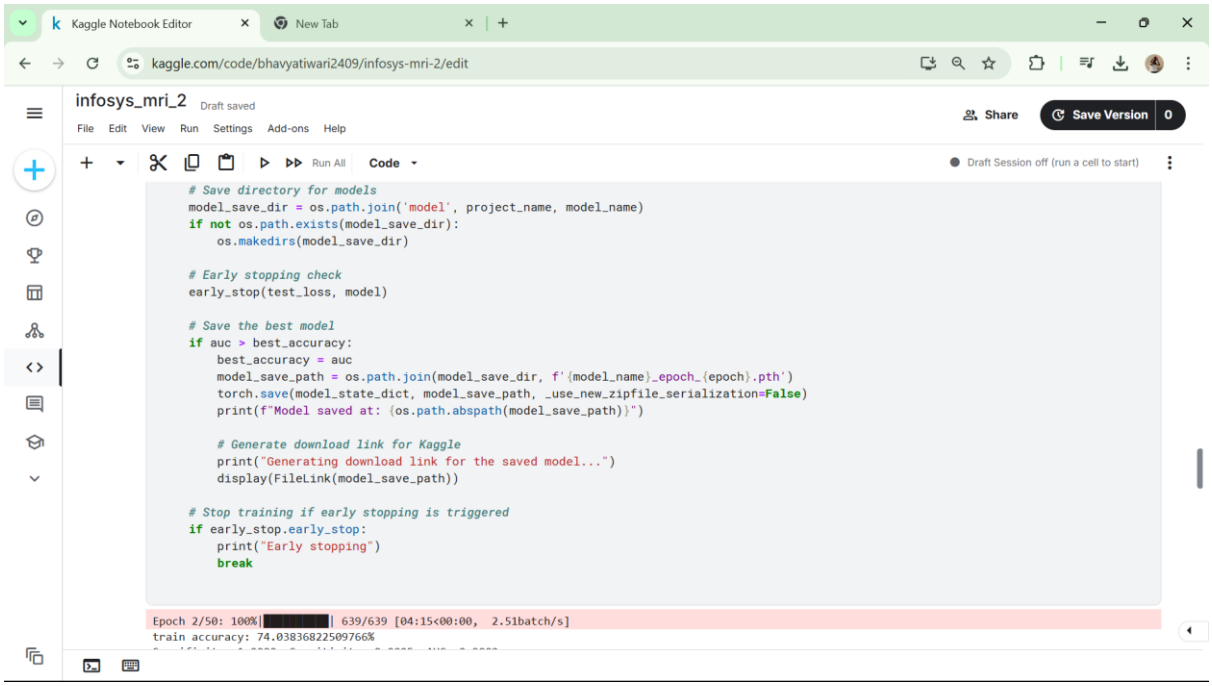


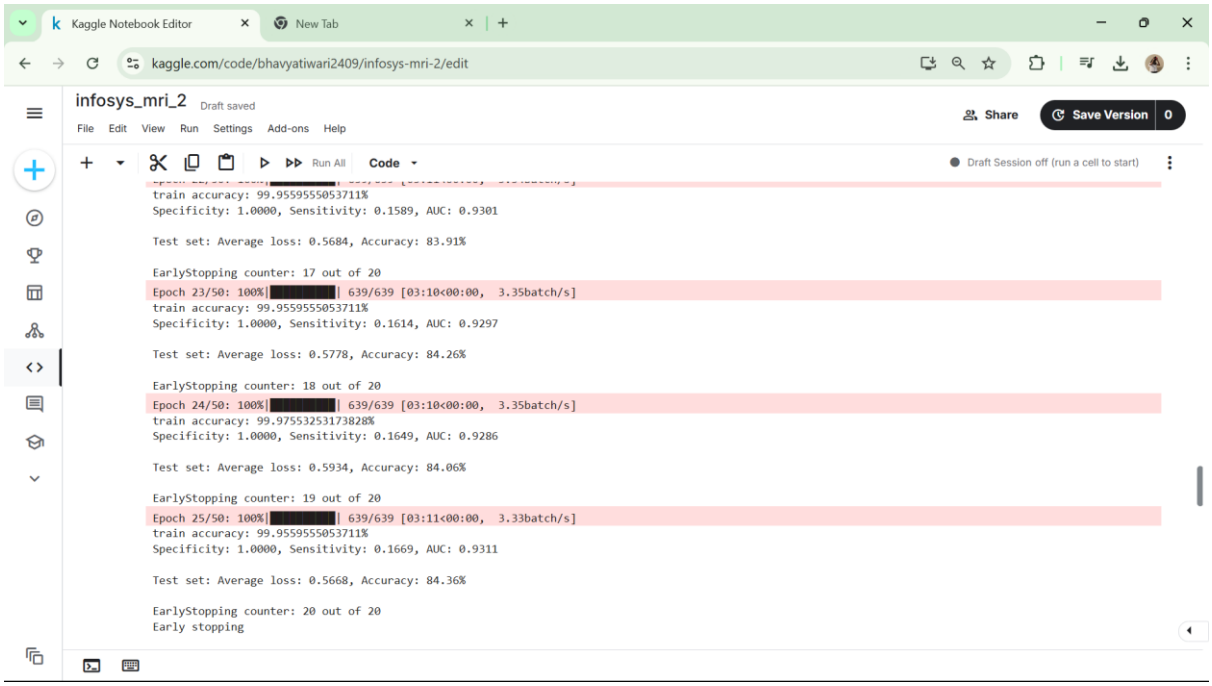

Kaggle Notebook Editor
New Tab
kaggle.com/code/bhavyatiwari2409/infosys-mri-2/edit
infosys_mri_2
Draft saved
File Edit View Run Settings Add-ons Help
Share
Save Version 0
Run All
Code
Draft Session off (run a cell to start)
for data, target in val_loader:
if torch.cuda.is_available():
data, target = data.cuda(), target.cuda()
val_output = model(data)
# Calculate test loss
test_loss += F.nll_loss(F.log_softmax(val_output, dim=1), target, reduction='sum').item()
# Get predictions and accumulate them
pred = val_output.data.max(1)[1]
all_predictions.extend(pred.cpu().numpy()) # Collect all predictions
all_targets.extend(target.cpu().numpy()) # Collect all target labels
# Calculate probabilities for AUC
possibility = F.softmax(val_output, dim=1).cpu().detach().numpy()
if possibilities is None:
possibilities = possibility
else:
possibilities = np.concatenate((possibilities, possibility), axis=0)
# Calculate the number of correct predictions
correct += pred.eq(target.data.view_as(pred)).cpu().sum()
# Compute confusion matrix
cm = metrics.confusion_matrix(all_targets, all_predictions)


Kaggle Notebook Editor
New Tab
kaggle.com/code/bhavyatiwari2409/infosys-mri-2/edit
infosys_mri_2
Draft saved
File Edit View Run Settings Add-ons Help
Share
Save Version 0
Run All
Code
Draft Session off (run a cell to start)
# One-hot encode the labels for AUC computation
num_classes = val_output.shape[1]
label_onehot = np.eye(num_classes)[np.array(all_targets).astype(int)]
# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(label_onehot.ravel(), possibilities.ravel())
auc_score = compute_auc(fpr, tpr) # Use 'compute_auc' to avoid conflicts
# Average test loss per sample
test_loss /= len(val_loader.dataset)
# Calculate specificity and sensitivity
specificity = 1 - fpr[1] if len(fpr) > 1 else 0
sensitivity = tpr[1] if len(tpr) > 1 else 0
print('Specificity: {:.4f}, Sensitivity: {:.4f}, AUC: {:.4f}'.format(specificity, sensitivity, auc_score))
print('\nTest set: Average loss: {:.4f}, Accuracy: {:.2f}%\n'.format(test_loss, 100. * correct / len(val_loader.dataset)))
return test_loss, 100. * correct / len(val_loader.dataset), cm, auc_score
[28]:
total_epochs = 50
lr = 0.01

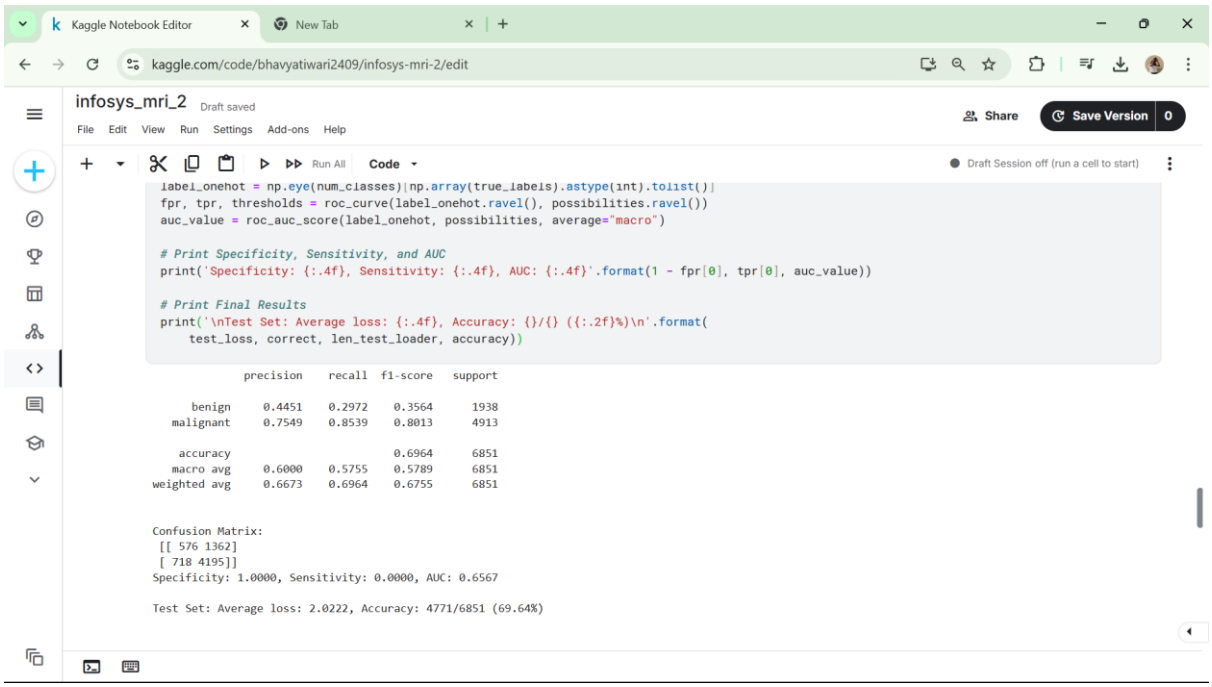
6. VGG-16 model training



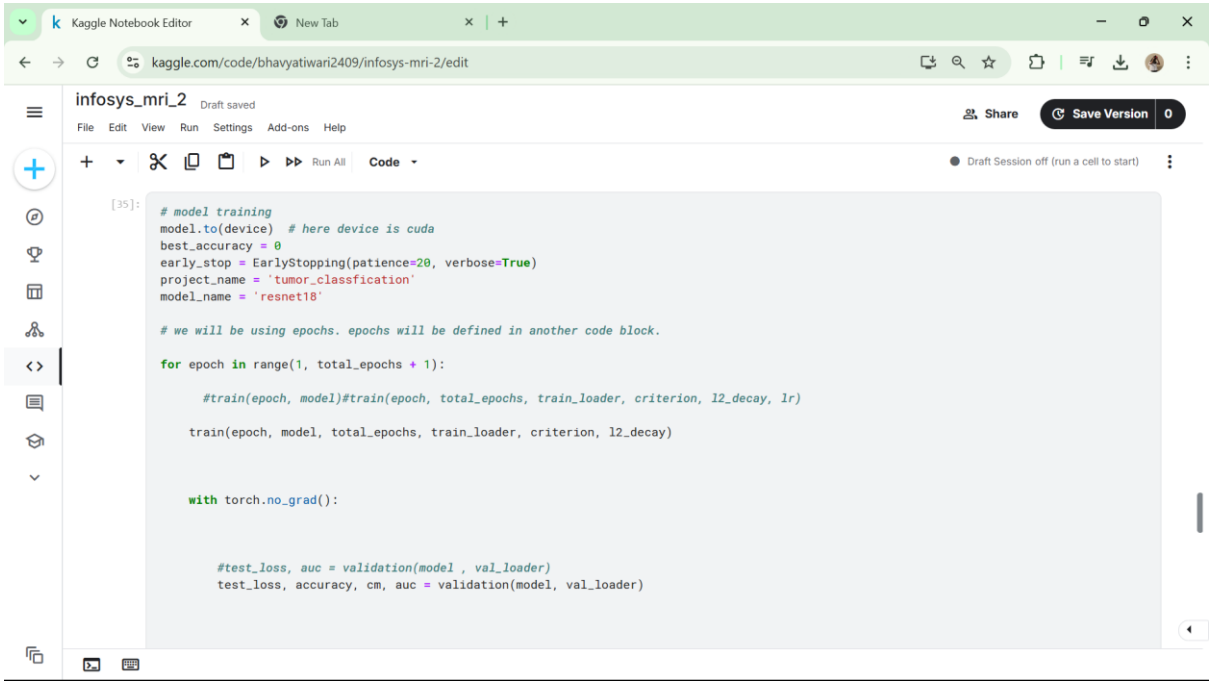


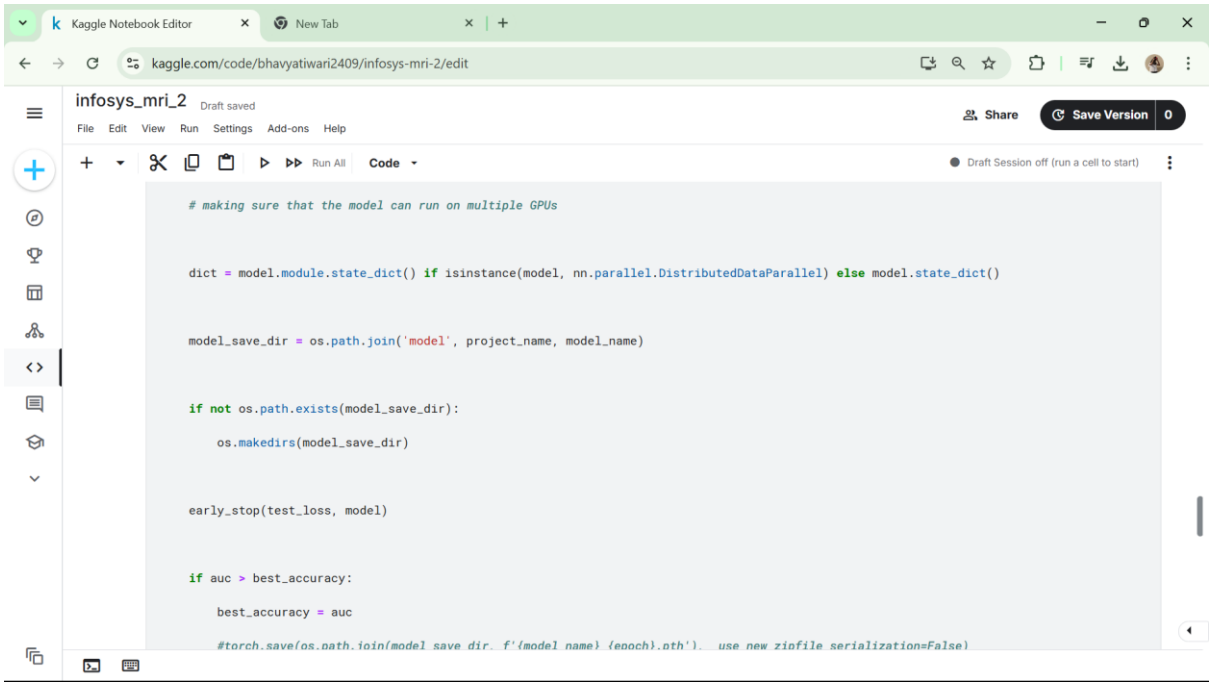


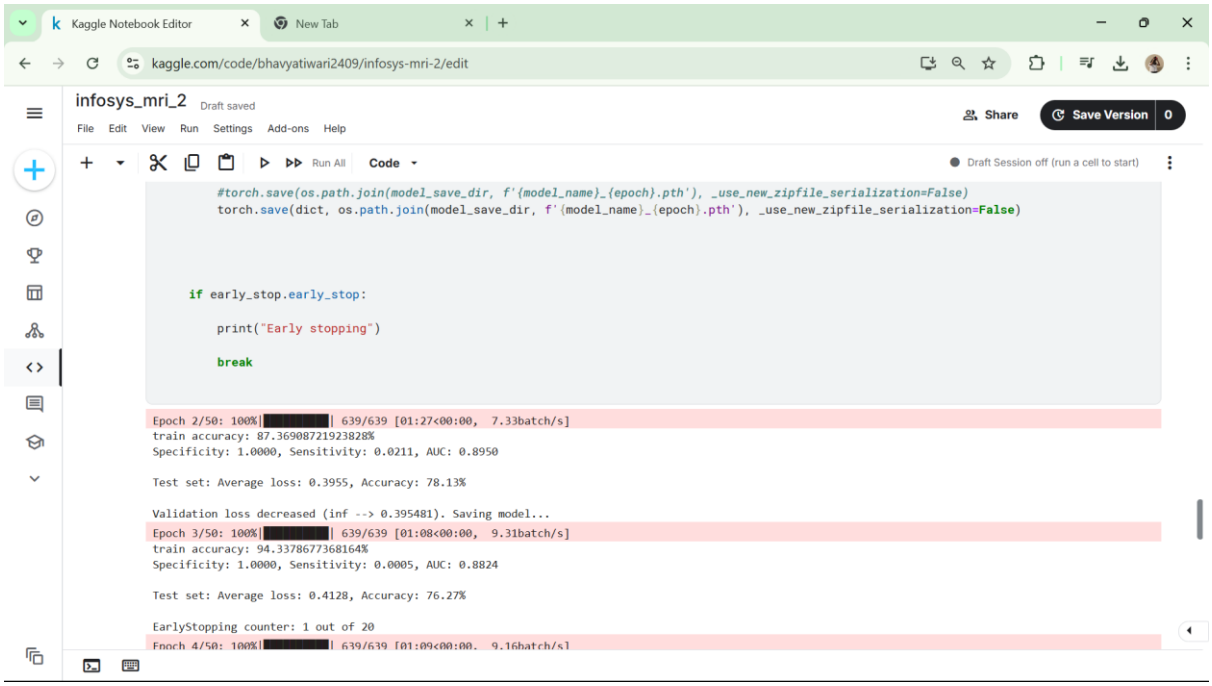
IMPROVEMENTS TO BE DONE: Handle overfitting

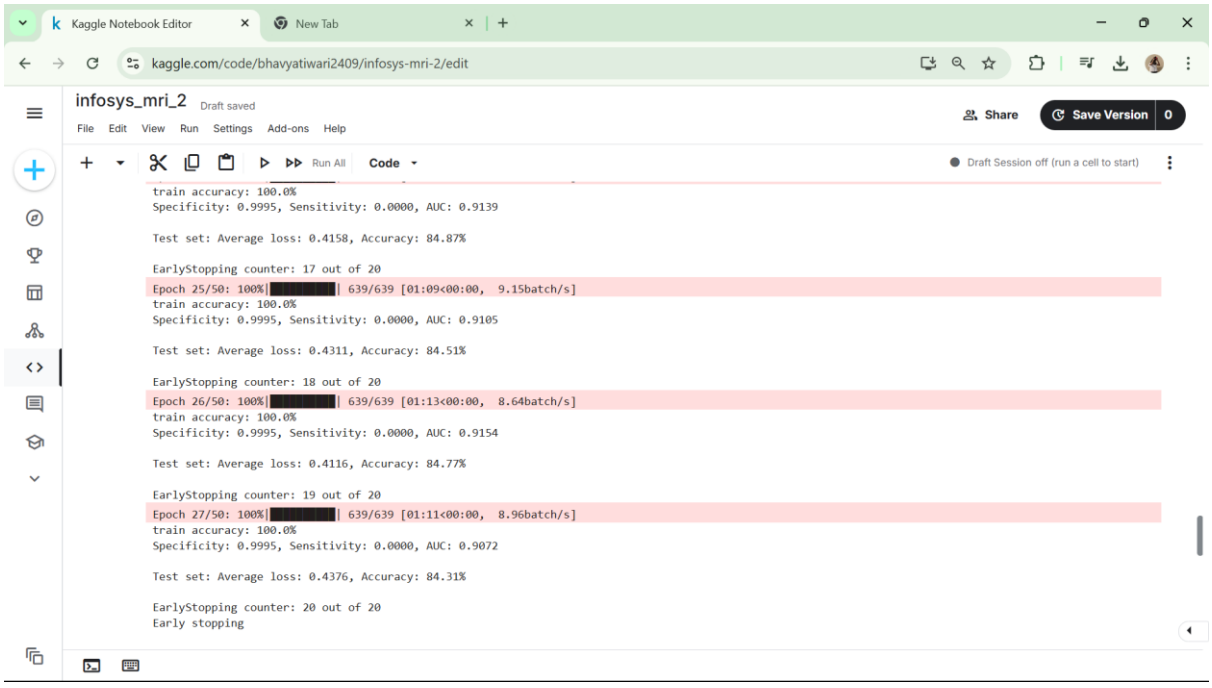


7. RESNET-18 training



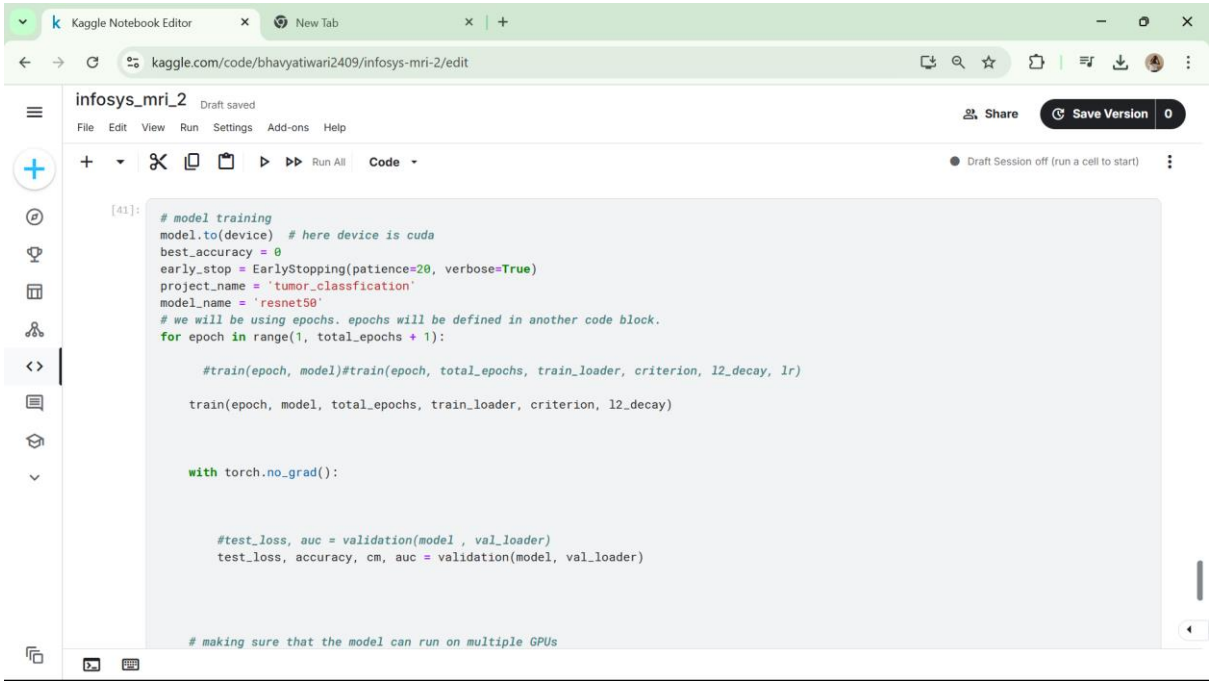


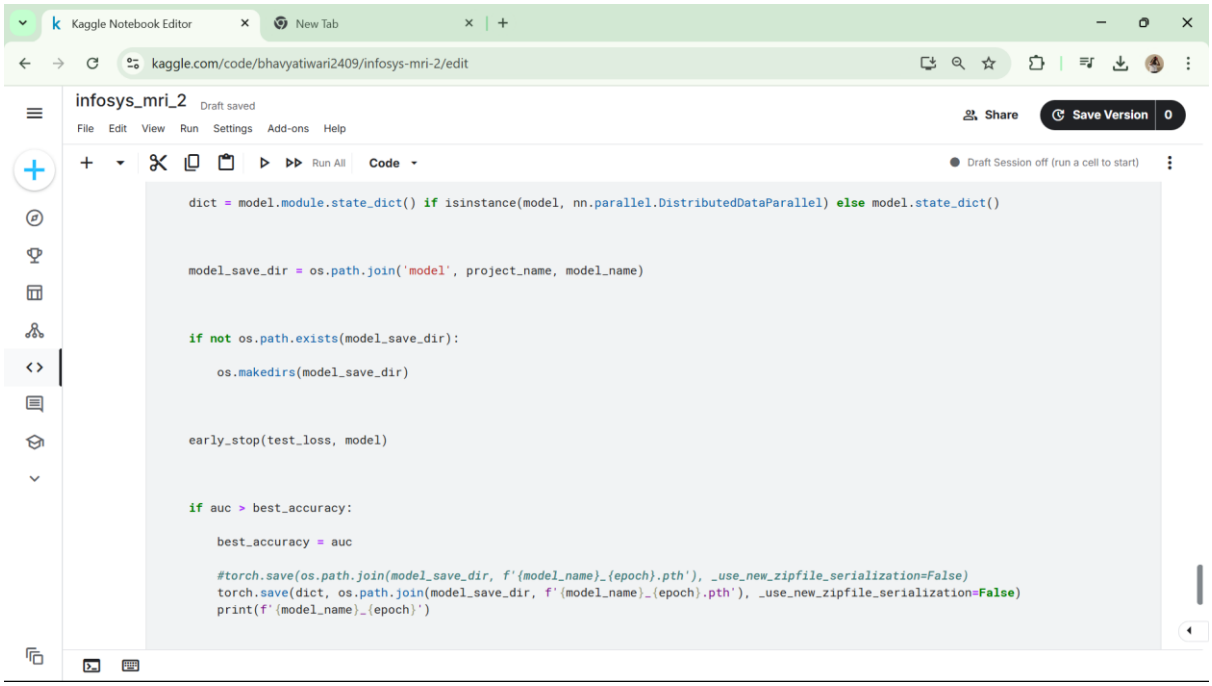


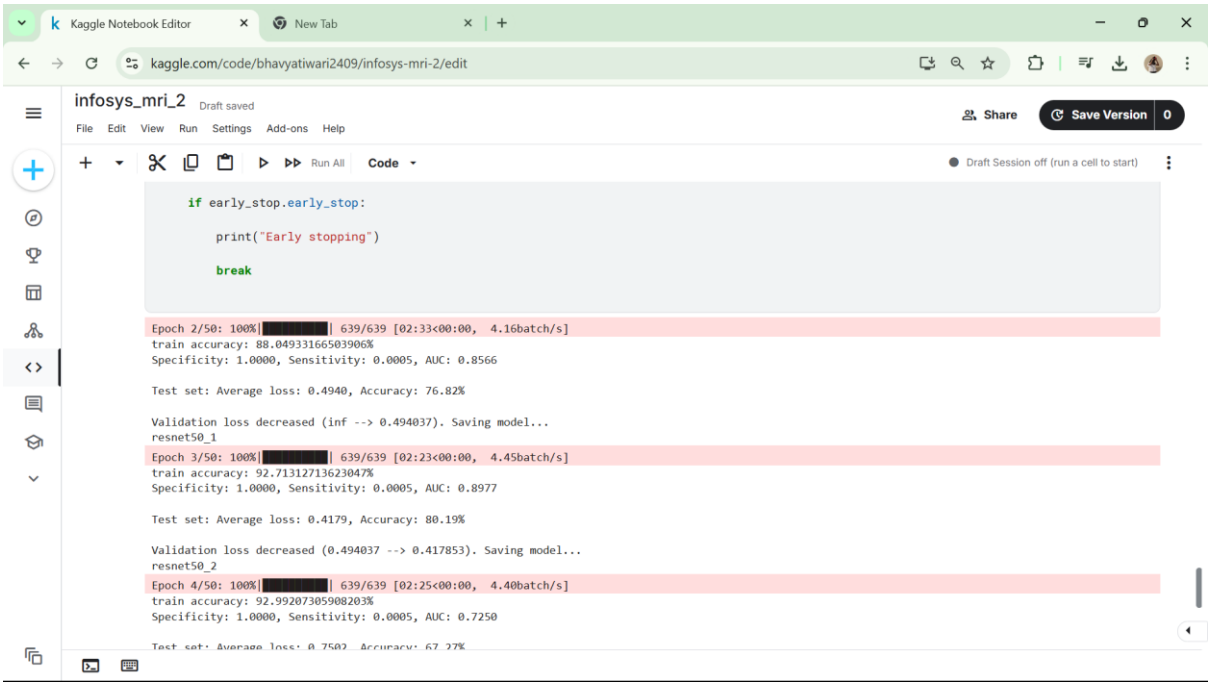


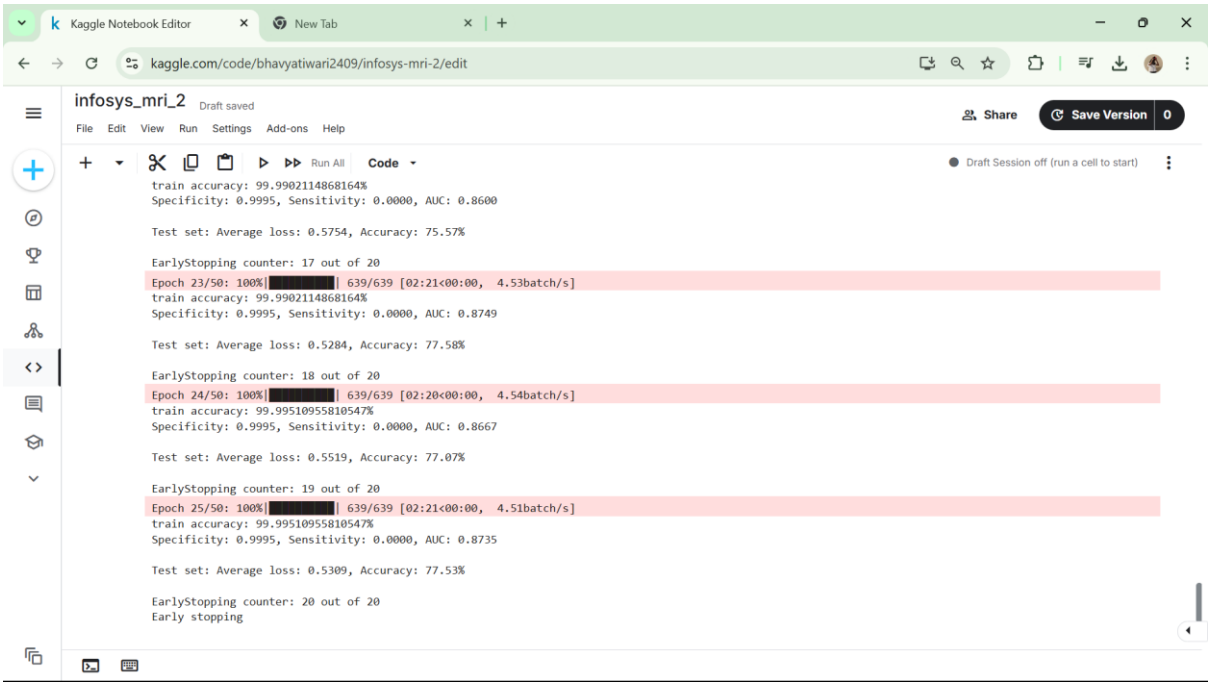
IMPROVEMENTS TO BE DONE: Handle overfitting

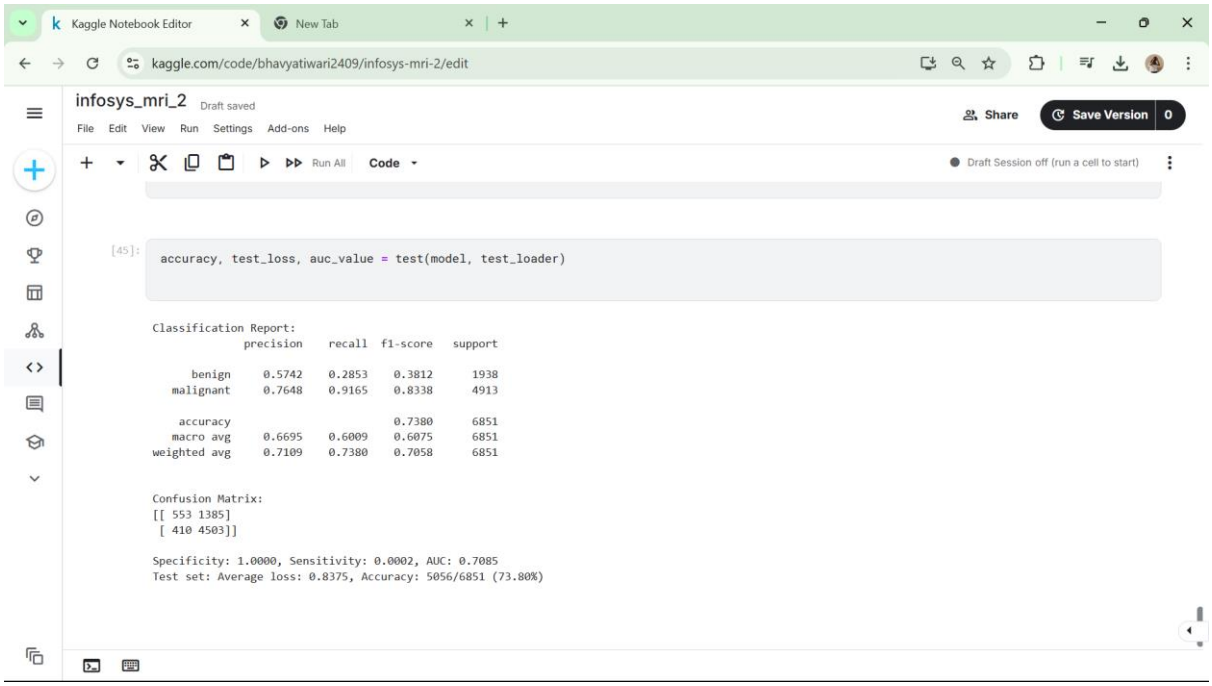
8. RESNET-50 Training






Kaggle Notebook Editor
New Tab
kaggle.com/code/bhavyatiwari2409/infosys-mri-2/edit
infosys_mri_2
Draft saved
File Edit View Run Settings Add-ons Help
Share
Save Version 0
Run All
Code
Draft Session off (run a cell to start)
if early_stop.early_stop:
print("Early stopping")
break
Epoch 2/50: 100%| | 639/639 [02:33<00:00, 4.16batch/s]
train accuracy: 88.04933166503906%
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.8566
Test set: Average loss: 0.4940, Accuracy: 76.82%
Validation loss decreased (inf --> 0.494037). Saving model...
resnet50_1
Epoch 3/50: 100%| | 639/639 [02:23<00:00, 4.45batch/s]
train accuracy: 92.71312713623047%
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.8977
Test set: Average loss: 0.4179, Accuracy: 80.19%
Validation loss decreased (0.494037 --> 0.417853). Saving model...
resnet50_2
Epoch 4/50: 100%| | 639/639 [02:25<00:00, 4.40batch/s]
train accuracy: 92.99207305908203%
Specificity: 1.0000, Sensitivity: 0.0005, AUC: 0.7250


Kaggle Notebook Editor
New Tab
kaggle.com/code/bhavyatiwari2409/infosys-mri-2/edit
infosys_mri_2
Draft saved
File Edit View Run Settings Add-ons Help
Share
Save Version 0
Run All
Code
Draft Session off (run a cell to start)
train accuracy: 99.9902114868164%
Specificity: 0.9995, Sensitivity: 0.0000, AUC: 0.8600
Test set: Average loss: 0.5754, Accuracy: 75.57%
EarlyStopping counter: 17 out of 20
Epoch 23/50: 100%| | 639/639 [02:21<00:00, 4.53batch/s]
train accuracy: 99.9902114868164%
Specificity: 0.9995, Sensitivity: 0.0000, AUC: 0.8749
Test set: Average loss: 0.5284, Accuracy: 77.58%
EarlyStopping counter: 18 out of 20
Epoch 24/50: 100%| | 639/639 [02:20<00:00, 4.54batch/s]
train accuracy: 99.99510955810547%
Specificity: 0.9995, Sensitivity: 0.0000, AUC: 0.8667
Test set: Average loss: 0.5519, Accuracy: 77.07%
EarlyStopping counter: 19 out of 20
Epoch 25/50: 100%| | 639/639 [02:21<00:00, 4.51batch/s]
train accuracy: 99.99510955810547%
Specificity: 0.9995, Sensitivity: 0.0000, AUC: 0.8735
Test set: Average loss: 0.5309, Accuracy: 77.53%
EarlyStopping counter: 20 out of 20
Early stopping



9. **Updates in code, changing hyperparameters and adding dropout layers**
10. **Running live demo on gradio**