Configuration Report

1. Introduction

1.1. Objective

The primary goal of this project is to leverage Machine Learning (ML) and Deep Learning (DL) techniques for breast cancer diagnosis using two datasets: the Breast Cancer Wisconsin (Diagnostic) dataset and the BreakHis Image dataset. The project involves preprocessing the datasets, training multiple ML and DL models, hyperparameter tuning, and evaluating their performance to identify the best-performing models for binary and multiclass classification tasks.

The document provides step-by-step instructions for running the code, including setting up the environment, preprocessing the data, training the models, evaluating them using test data, and generating results.

2. Hardware and Software Configurations

2.1 Hardware

- Processor: 12th Gen Intel(R) Core(TM) i5-1235U @ 1.30 GHz
- Installed RAM: 16.0 GB (15.7 GB usable)
- System Type: 64-bit Operating System, x64-based Processor
- Storage: 512 GB SSD

2.2 Software

- Operating System: Windows 11
- Programming Language: Python 3.10
- IDE: Jupyter Notebook (Anaconda Distribution)
- Libraries:

3. Installing Modules and Libraries

Libraries that need to be installed are:

!pip install numpy; pandas; matplotlib; seaborn; sklearn; tensorflow; ucimlrepo

Then import the following libraries:

```
•[1]: # Import Necessary Libraries
import numpy as np
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import precision_score, recall_score, f1_score
from tensorflow.keras import layers, models, optimizers, callbacks, regularizers
import tensorflow as tf
from tensorflow.keras.callbacks import ModelCheckpoint
import os
import shutil
from glob import glob
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import cv2
import random
import matplotlib.image as mpimg
import pandas as pd
from ucimlrepo import fetch_ucirepo
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import RFE, SelectFromModel, mutual_info_classif
from sklearn.ensemble import RandomForestClassifier
import seaborn as sns
from sklearn.linear_model import LassoCV
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score, cross_validate, StratifiedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import make_scorer, accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, roc_curve, confusion_matrix, classifica
from sklearn.preprocessing import StandardScaler
from xgboost import XGBClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import SGDClassifier
from sklearn.tree import DecisionTreeClassifier
import plotly.graph_objects as go
from plotly.subplots import make_subplots
from ucimlrepo import fetch_ucirepo
```

4. Step-by-Step Workflow

4.1 Data Preparation

4.1.1 Wisconsin Dataset

Download and Load Data:

The **Breast Cancer Wisconsin (Diagnostic) Dataset** was downloaded from the UCI Machine Learning Repository. The dataset is provided in .csv format and contains 569 records with 32 columns, including one target column (diagnosis) and 31 numerical features.

Run the code mentioned in Fig 2 to import the dataset:

```
!pip install ucimlrepo
```

```
from ucimlrepo import fetch_ucirepo

# fetch dataset
breast_cancer_wisconsin_diagnostic = fetch_ucirepo(id=17)

# data (as pandas dataframes)
X = breast_cancer_wisconsin_diagnostic.data.features
y = breast_cancer_wisconsin_diagnostic.data.targets
```

Figure 2: Import Wisconsin .csv dataset

Preprocessing:

1. Check for any Missing Values (null)
2. Feature Extraction

   Select the top features for the machine learning models, to reduce the dimensionality from 30 features. Following feature selection methods have been carried out:

- Correlation: is fast and easy but limited to linear relationships.

```
# Plotting the correlation matrix
plt.figure(figsize=(12, 8))
correlation_matrix = pd.DataFrame(X).assign(Target=y).corr()
sns.heatmap(correlation_matrix, annot=False, cmap='coolwarm')
plt.title("Feature Correlation Matrix with Target")
plt.show()
```

```
# Extract Best Correlated Features
threshold = 0.7
best_features_corr = correlation_with_target[abs(correlation_with_target) > threshold].index.tolist()
print(f"Best correlated features with the target (threshold = {threshold}): {best_features_corr}")
```

Figure 3: Feature Selection - Correlation

- RFE: works well for recursive elimination but can be slow on large datasets.

```
estimator = RandomForestClassifier(random_state=42)
rfe_selector = RFE(estimator, n_features_to_select=8)
rfe_selector.fit(X_scaled, y)
best_features_rfe = X.columns[rfe_selector.support_]
print("RFE selected features:", best_features_rfe)
```

Figure 4: Feature Selection - Recursive Feature Elimination

- Random Forest: captures non-linear interactions.

```
# Feature Importance from Random Forest
forest = RandomForestClassifier(random_state=42)
forest.fit(X_scaled, y)
importances = forest.feature_importances_
best_features_forest = X.columns[np.argsort(importances)[-8:]]  # Top 10 features
print("Random Forest selected features:", best_features_forest)
```

Figure 5: Feature Selection – Feature Importance from Random Forest

- Lasso: performs regularization and selects fewer but important features.

```
# Lasso (L1 Regularization)
lasso = LassoCV(cv=5, max_iter=10000).fit(X_scaled, y)
best_features_lasso = X.columns[lasso.coef_ != 0]  # Features with non-zero coefficients
print("Lasso selected features:", best_features_lasso)
```

Figure 6: Feature Selection – Lasso Feature Selection

- Mutual Information: captures non-linear dependencies not handled by traditional correlation metrics.

```
# 6. Mutual Information
mutual_info = mutual_info_classif(X_scaled, y)
best_features_mutual_info = X.columns[np.argsort(mutual_info)[-8:]]  # Top 10 features
print("Mutual Information selected features:", best_features_mutual_info)
```

Figure 7: Feature Selection – Mutual Information

The selected features were a combined as per Fig 8:

```
# Summarizing all selected features
selected_features = set(best_features_corr) | set(best_features_rfe) | set(best_features_forest) | set(best_features_mutual_info)
print("\nCombined selected features from all methods:", selected_features)
```

```
Combined selected features from all methods: {'perimeter1', 'perimeter3', 'radius1', 'concave_points3', 'area3', 'concavity1', 'concave_points1', 'radius
3', 'area1', 'texture3'}
```

Figure 8: Selected Features

3. Encoded the independent variable (B:0, M:1)
4. Standardize the data (Fig 9):

```
# Standardize the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_eval_scaled = scaler.transform(X_eval)
```

Figure 9: Standardize data

5. Splitting the Data

   The dataset was split into training, validation and testing subsets using a 60-20-20 split ratio (as per Fig 10) while ensuring class balance with stratification:

```
# 1. Train-Test-Eval Split (60% train, 20% eval, 20% test)
selected_features = list(selected_features) # Convert from set to a list for usage
X_train_eval, X_test, y_train_eval, y_test = train_test_split(X[selected_features], y, test_size=0.2, random_state=42, stratify=y)
X_train, X_eval, y_train, y_eval = train_test_split(X_train_eval, y_train_eval, test_size=0.25, random_state=42, stratify=y_train_eval)
```

Figure 10: Dataset Split

4.1.2 BreakHis Dataset

Dataset Description:

The BreakHis dataset is a medical image dataset widely used for classifying breast cancer histopathological images. It is structured as follows:

- **Binary Classification**: Images are divided into **benign** and **malignant** categories.
- **Multiclass Classification**: Malignant and Benign images are further categorized into eight distinct cancer types:
    1. Adenosis
    2. Fibroadenoma
    3. Tubular Adenoma
    4. Phyllodes Tumor
    5. Ductal Carcinoma
    6. Lobular Carcinoma
    7. Mucinous Carcinoma
    8. Papillary Carcinoma

The dataset provides a comprehensive base for both binary and multiclass classification tasks, with the challenge of handling class imbalances.

Download and Load Dataset

The Breast Cancer Histopathological Database (BreakHis) can be accessed from the [Laboratório Visão Robótica e Imagem]() website and alternatively, can be downloaded from [BreakHis-Kaggle](). The dataset is composed of 9,109 microscopic images of breast tumor tissue collected from 82 patients using different magnifying factors (40X, 100X, 200X, and 400X). Load the dataset by running the code in Fig 11.

```python
# Base directory path for the BreakHis dataset
base_dir = 'archive/BreaKHis_v1/BreaKHis_v1/histology_slides/breast/'

# Dictionary to hold file paths for benign and malignant types
image_paths = {
    'benign': {},
    'malignant': {}
}

# Function to load image paths, considering the detailed directory structure
def load_image_paths(base_dir, classes=('benign', 'malignant')):
    for tumor_class in classes:
        class_path = os.path.join(base_dir, tumor_class, 'SOB')  # Navigate to the SOB folder within each class

        # Traverse each tumor type (e.g., adenosis, ductal_carcinoma)
        for tumor_type in os.listdir(class_path):
            tumor_type_path = os.path.join(class_path, tumor_type)

            # Initialize dictionary to store images by magnification for each type
            image_paths[tumor_class][tumor_type] = {}

            # Traverse each patient's folder within the tumor type
            for patient_folder in os.listdir(tumor_type_path):
                patient_path = os.path.join(tumor_type_path, patient_folder)

                if os.path.isdir(patient_path):
                    # Traverse each magnification level folder within the patient folder
                    for magnification in os.listdir(patient_path):
                        magnification_path = os.path.join(patient_path, magnification)

                        # Get list of image files in the magnification folder
                        images = [
                            os.path.join(magnification_path, img)
                            for img in os.listdir(magnification_path)
                            if img.endswith('.png')
                        ]

                        # Store image paths under the tumor type and magnification level
                        if magnification not in image_paths[tumor_class][tumor_type]:
                            image_paths[tumor_class][tumor_type][magnification] = []

                        # Append images to the list for the specific magnification level
                        image_paths[tumor_class][tumor_type][magnification].extend(images)

    print("Image paths loaded successfully.")
    return image_paths

# Load the image paths for benign and malignant classes
image_paths = load_image_paths(base_dir)
```

Figure 11: Import BreakHis dataset

Visualization:

Class distributions were analyzed to understand data imbalance:

- **Binary Classification**:

The total count of benign and malignant images was visualized using a bar plot to highlight the data skew.

```
# Dictionary to store counts for each subclass and overall benign/malignant counts
counts = {'benign': {}, 'malignant': {}, 'total_benign': 0, 'total_malignant': 0}

# Iterate through each tumor class and type in the image_paths
for tumor_class, types_dict in image_paths.items():
    for tumor_type, magnifications_dict in types_dict.items():
        # Calculate the total number of images for each tumor type across magnifications
        total_images = sum(len(images) for images in magnifications_dict.values())

        # Store the count for each subclass
        counts[tumor_class][tumor_type] = total_images

        # Add to the total benign or malignant count
        if tumor_class == 'benign':
            counts['total_benign'] += total_images
        elif tumor_class == 'malignant':
            counts['total_malignant'] += total_images
```

```
# Create the plot
plt.figure(figsize=(8, 2))
plt.barh(list(counts.keys()), list(counts.values()), color=['skyblue', 'salmon'])
plt.xlabel("Number of Images")
plt.title("Distribution of Images in Benign and Malignant Classes")
plt.show()
```

Figure 12: Visualize Binary Class Distribution

- **Multiclass Classification**:

The counts of images for each of the eight cancer types were plotted to reveal imbalances across classes.

```
# Extract the subclass names and their counts
benign_subclasses = list(counts['benign'].keys())
benign_counts = list(counts['benign'].values())
malignant_subclasses = list(counts['malignant'].keys())
malignant_counts = list(counts['malignant'].values())

# Set up the figure
plt.figure(figsize=(10, 6))

# Plot the benign and malignant subclass distributions
plt.bar(benign_subclasses, benign_counts, color='skyblue', label='Benign', alpha=0.7)
plt.bar(malignant_subclasses, malignant_counts, color='salmon', label='Malignant', alpha=0.7)

# Add labels and title
plt.xlabel('Tumor Subclass')
plt.ylabel('Number of Images')
plt.title('Distribution of Images in Benign and Malignant Subclasses')
plt.xticks(rotation=45, ha='right')  # Rotate x-axis labels for better readability

# Place legend outside of the plot area
plt.legend(loc='upper left', bbox_to_anchor=(1, 1))

# Adjust layout for readability
plt.tight_layout()

# Show the plot
plt.show()
```

Figure 13: Visualize Multi Class Distribution

These visualizations helped identify augmentation needs to balance the dataset.

Augmentation:

Binary Classification

To balance the dataset for binary classification (refer fig 14):

- A combination of image augmentation techniques such as **rotation**, **horizontal/vertical flipping**, **cropping**, and **zooming** was applied.

- The original dataset of 7,909 images was expanded to **20,778 images** by merging the original and augmented images.
- This augmentation ensured the counts of benign and malignant images were approximately equal.

```python
# Set parameters
IMG_SIZE = (224, 224)  # Resize to 224x224
benign_aug_count = 3  # Apply augmentation 3 times on each benign image
malignant_aug_count = 1  # Apply augmentation once on each malignant image

# Augmentation setup
datagen_benign = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    fill_mode='nearest'
)

datagen_malignant = ImageDataGenerator(
    rotation_range=10,
    zoom_range=0.1,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

```python
# Function to load and resize images from the image paths
def load_and_resize_images(image_paths, img_size):
    images = []
    labels = []
    for tumor_class, tumor_data in image_paths.items():
        for tumor_type, magnification_data in tumor_data.items():
            for magnification, img_paths in magnification_data.items():
                for img_path in img_paths:
                    img = cv2.imread(img_path)
                    img = cv2.resize(img, img_size)  # Resize image to target size
                    images.append(img)
                    labels.append(f'{tumor_class}_{tumor_type}')
    return np.array(images), np.array(labels)

# Load and resize images from the paths
images, labels = load_and_resize_images(image_paths, IMG_SIZE)
```

```python
# Function to save images with augmentations in batches
def augment_and_save_images(image_paths, output_dir, img_size, benign_aug_count, malignant_aug_count):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    for tumor_class, tumor_data in image_paths.items():
        # Set augmentation generator and count
        if tumor_class == 'benign':
            datagen = datagen_benign
            aug_count = benign_aug_count
        else:
            datagen = datagen_malignant
            aug_count = malignant_aug_count

        print(f"\nProcessing tumor class: {tumor_class.upper()}")  # Progress print

        # Iterate over tumor types and magnifications
        for tumor_type, magnifications in tumor_data.items():
            print(f"  Tumor type: {tumor_type}")  # Progress print

            for magnification, img_files in magnifications.items():
                print(f"    Magnification level: {magnification} - Total images: {len(img_files)}")  # Progress print

                for idx, img_path in enumerate(img_files):
                    img = cv2.imread(img_path)
                    img = cv2.resize(img, img_size)  # Resize to target size

                    # Reshape for augmentation
                    img = img.reshape((1,) + img.shape)

                    # Create augmented images and save to disk
                    for i, batch in enumerate(datagen.flow(img, batch_size=1)):
                        aug_img = batch[0].astype(np.uint8)

                        # Create folder structure for saving augmented images
                        aug_output_dir = os.path.join(output_dir, tumor_class, tumor_type, magnification)
                        os.makedirs(aug_output_dir, exist_ok=True)

                        # Save the augmented image
                        aug_img_filename = f"{os.path.splitext(os.path.basename(img_path))[0]}_aug_{i}.png"
                        cv2.imwrite(os.path.join(aug_output_dir, aug_img_filename), aug_img)

                        # Progress print for each augmented image
                        print(f"      Saved augmentation {i + 1}/{aug_count} for image {idx + 1}/{len(img_files)} in {tumor_type} ({magnification})")

                        # Stop after generating the specified number of augmentations
                        if i >= aug_count - 1:
                            break

    print(f"\nAugmentation completed. Augmented images saved to {output_dir}")

# Set the output directory for augmented images
output_dir = 'archive/BreaKHis_v1/BreaKHis_v1/histology_slides/breast/augmented_images_binary'

# Load image paths and process images in batches for augmentation
image_paths = load_image_paths(base_dir)
augment_and_save_images(image_paths, output_dir, IMG_SIZE, benign_aug_count, malignant_aug_count)
```

Figure 14: Data Augmentation for Binary Class

Multiclass Classification

For multiclass classification:

- **Target Size Calculation**: The class with the highest count, **Ductal Carcinoma** (~3,500 images), was set as the target size.

```
sub_class_target_image_count = 3500  # Target image count per class after augmentation
```

```
# Function to count and balance images per class and magnification
def calculate_augmentation_counts(image_paths, target_image_count):
    augmentation_counts = {}

    # Iterate through each tumor class, tumor type, and magnification
    for tumor_class, tumor_data in image_paths.items():
        augmentation_counts[tumor_class] = {}

        for tumor_type, magnifications in tumor_data.items():
            total_image_count = 0

            # Sum the images across all magnification levels for this tumor type
            for magnification, img_files in magnifications.items():
                total_image_count += len(img_files)  # Count the images in this magnification
                print(f"Class: {tumor_class}, Type: {tumor_type}, Magnification: {magnification}, Image Count: {len(img_files)}")

            # Calculate how many augmentations are needed for this tumor type to meet the target
            print(f"Total image count for {tumor_class} - {tumor_type}: {total_image_count}")

            # Calculate how many augmentations are needed to meet the target
            if total_image_count <= target_image_count:
                aug_count_needed = (target_image_count - total_image_count) // total_image_count
            else:
                aug_count_needed = 0  # No augmentations needed if the count already exceeds the target

            augmentation_counts[tumor_class][tumor_type] = aug_count_needed

    return augmentation_counts

# Load image path
image_paths = load_image_paths(base_dir)
# Calculate augmentation counts based on the current class distribution
augmentation_counts = calculate_augmentation_counts(image_paths, sub_class_target_image_count)
```

Figure 15: Multi Class – Augmentation Count

- **Augmentation Factors**:
  - Each class was analyzed, and an augmentation factor was calculated to scale the count of images to match the target size.
  - For example, if a class had 1,000 images, a factor of 3.5 was applied to generate ~3,500 images.

After augmentation, all eight classes had a similar image count (~3,000+ images), achieving a balanced dataset.

```
# Initialize the augmentation generators (adjust these based on your requirements)
datagen_benign = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

datagen_malignant = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

```python
# Function to save images with augmentations in batches
def augment_and_save_images(image_paths, output_dir, img_size, augmentation_counts):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    for tumor_class, tumor_data in image_paths.items():
        # Set augmentation generator and count
        if tumor_class == 'benign':
            datagen = datagen_benign
            aug_count_dict = augmentation_counts['benign']
        else:
            datagen = datagen_malignant
            aug_count_dict = augmentation_counts['malignant']

        print(f"\nProcessing tumor class: {tumor_class.upper()}")  # Progress print

        # Iterate over tumor types and magnifications
        for tumor_type, magnifications in tumor_data.items():
            print(f"  Tumor type: {tumor_type}")  # Progress print

            for magnification, img_files in magnifications.items():
                print(f"    Magnification level: {magnification} - Total images: {len(img_files)}")  # Progress print

                # Get the augmentation count for the current tumor type
                aug_count = aug_count_dict.get(tumor_type, 0)
                if aug_count == 0:
                    print(f"      No augmentation needed for {tumor_type} in {magnification}. Skipping.")
                    continue

                for idx, img_path in enumerate(img_files):
                    img = cv2.imread(img_path)
                    img = cv2.resize(img, img_size)  # Resize to target size

                    # Reshape for augmentation
                    img = img.reshape((1,) + img.shape)

                    # Create augmented images and save to disk
                    for i, batch in enumerate(datagen.flow(img, batch_size=1)):
                        aug_img = batch[0].astype(np.uint8)

                        # Create folder structure for saving augmented images
                        aug_output_dir = os.path.join(output_dir, tumor_class, tumor_type, magnification)
                        os.makedirs(aug_output_dir, exist_ok=True)

                        # Save the augmented image
                        aug_img_filename = f"{os.path.splitext(os.path.basename(img_path))[0]}_aug_{i}.png"
                        cv2.imwrite(os.path.join(aug_output_dir, aug_img_filename), aug_img)

                        # Progress print for each augmented image
                        print(f"      Saved augmentation {i + 1}/{aug_count} for image {idx + 1}/{len(img_files)} in {tumor_type} ({magnification})"

                        # Stop after generating the specified number of augmentations
                        if i >= aug_count - 1:
                            break

    print(f"\nAugmentation completed. Augmented images saved to {output_dir}")

# Set the output directory for augmented images
output_dir = 'archive/BreaKHis_v1/BreaKHis_v1/histology_slides/breast/augmented_images_subclass/'

# Load image paths (assuming you have this function to load image paths)
image_paths = load_image_paths(base_dir)

# Perform data augmentation
augment_and_save_images(image_paths, output_dir, IMG_SIZE, augmentation_counts)
```

Figure 16: Multi Class – Data Augmentation

4.2 Tensor Preparation

Binary and Multiclass: Image Flattening and Tensor Conversion

To process the image datasets for both binary and multiclass classification, the image files were organized into respective class-specific directories. Each class was assigned a unique label to ensure seamless conversion into Tensor format.

Flattening the Directory:

- For binary classification, the original and augmented images of benign and malignant classes were merged into a single directory for each class (Fig 17).

```python
# Define source directories (original and augmented) and target directories
original_dir = base_dir
augmented_dir = output_dir  # where augmented images were saved
merged_dir = 'archive/BreaKHis_v1/BreaKHis_v1/histology_slides/breast/merged_images_binary'  # final directory for all images

# Ensure merged directories for benign and malignant
os.makedirs(os.path.join(merged_dir, 'benign'), exist_ok=True)
os.makedirs(os.path.join(merged_dir, 'malignant'), exist_ok=True)

# Helper function to copy images from original and augmented directories
def merge_images(src_dir, dest_dir):
    # Use glob to get all image files in source directory
    img_files = glob(os.path.join(src_dir, '**', '*.png'), recursive=True)
    for img_path in img_files:
        # Copy each image file to the destination directory
        shutil.copy(img_path, dest_dir)

# Merge original and augmented images for both benign and malignant classes
print("Merging benign images...")
merge_images(os.path.join(original_dir, 'benign'), os.path.join(merged_dir, 'benign'))
merge_images(os.path.join(augmented_dir, 'benign'), os.path.join(merged_dir, 'benign'))

print("Merging malignant images...")
merge_images(os.path.join(original_dir, 'malignant'), os.path.join(merged_dir, 'malignant'))
merge_images(os.path.join(augmented_dir, 'malignant'), os.path.join(merged_dir, 'malignant'))

# Output the final image count
total_benign = len(os.listdir(os.path.join(merged_dir, 'benign')))
total_malignant = len(os.listdir(os.path.join(merged_dir, 'malignant')))
total_images = total_benign + total_malignant

print("\nFinal image counts after merging:")
print(f"Total benign images: {total_benign}")
print(f"Total malignant images: {total_malignant}")
print(f"Total images: {total_images}")
```

Figure 17: Binary Class – Merge Original and Augmented Images

- For multiclass classification, the images from all eight classes (e.g., ductal carcinoma, lobular carcinoma, etc.) were similarly flattened into individual directories.

```python
def merge_images(image_paths, augmented_dir, output_dir):
    total_counts = {'benign': {}, 'malignant': {}}

    # Loop through the image paths for each tumor class and subtype
    for tumor_class, tumor_types in image_paths.items():
        print(f"\nMerging images for {tumor_class.upper()} class")

        for tumor_type, magnifications in tumor_types.items():
            output_type_dir = os.path.join(output_dir, tumor_class, tumor_type)
            os.makedirs(output_type_dir, exist_ok=True)

            total_count = 0  # Initialize count for this subclass

            # Loop through each magnification level
            for magnification, images in magnifications.items():
                output_magnification_dir = os.path.join(output_type_dir, magnification)
                os.makedirs(output_magnification_dir, exist_ok=True)

                # Copy original images from image_paths
                for img_path in images:
                    destination_path = os.path.join(output_magnification_dir, os.path.basename(img_path))
                    if not os.path.exists(destination_path):  # Only copy if not already present
                        shutil.copy(img_path, destination_path)
                        total_count += 1

                # Add augmented images if available
                augmented_magnification_dir = os.path.join(augmented_dir, tumor_class, tumor_type, magnification)
                if os.path.exists(augmented_magnification_dir):
                    for img_file in os.listdir(augmented_magnification_dir):
                        img_path = os.path.join(augmented_magnification_dir, img_file)
                        destination_path = os.path.join(output_magnification_dir, img_file)
                        if os.path.isfile(img_path) and not os.path.exists(destination_path):
                            shutil.copy(img_path, destination_path)
                            total_count += 1

            # Store total count for this subclass
            total_counts[tumor_class][tumor_type] = total_count
            print(f"  {tumor_type}: {total_count} images")

    return total_counts


# Directories
augmented_dir = 'archive/BreaKHis_v1/BreaKHis_v1/histology_slides/breast/augmented_images_subclass/'
merged_dir = 'archive/BreaKHis_v1/BreaKHis_v1/histology_slides/breast/merged_images_subclass'

# Merge images and calculate counts using the preloaded image_paths
total_image_counts = merge_images(image_paths, augmented_dir, merged_dir)
```

Figure 18: Multi Class – Merge Original and Augmented Images

- This structured approach ensured that the tensor processing package recognized the data hierarchy correctly.

Tensor Conversion:

- The images were loaded as NumPy arrays using a custom preprocessing pipeline and converted into tensors using TensorFlow's image_dataset_from_directory() function. This method automatically assigned labels based on the directory structure.
- The images were resized to a standard size (e.g., 224x224 pixels) to maintain consistency.

Dataset Splitting: Train (70%), Validation (15%), Test (15%)

After the tensor conversion, the dataset was split into three subsets to ensure robust training and evaluation of the models. The splitting process ensured class balance by using the stratified split method. The split ratios were consistent across both binary and multiclass datasets to maintain uniformity in the evaluation process (Fig 19).

```python
def load_and_split_dataset(directory, img_size=(224, 224), batch_size=32, seed=123):
    """
    Load images from the given directory and split into training, validation, and test sets.

    Parameters:
    - directory: Path to the images directory.
    - img_size: Target image size.
    - batch_size: Number of samples per batch.
    - seed: Random seed for reproducibility.

    Returns:
    - train_ds: Training dataset.
    - val_ds: Validation dataset.
    - test_ds: Test dataset.
    """

    # Determine label mode based on dataset type (binary or multiclass)
    # int for binary, categorical for multiclass
    label_mode='int' if "binary" in directory else 'categorical'

    # Load the full dataset from directory
    full_ds = tf.keras.preprocessing.image_dataset_from_directory(
        directory,
        image_size=img_size,
        batch_size=batch_size,
        seed=seed,
        label_mode=label_mode  # Adjust label mode for binary or multiclass
    )

    # Convert binary labels to required format if necessary
    if label_mode == 'int':
        # Map to ensure binary labels are in shape (batch_size, 1) by expanding dimensions
        full_ds = full_ds.map(lambda x, y: (x, tf.expand_dims(y, axis=-1)))

    # Shuffle the dataset first
    dataset_size = len(full_ds)
    full_ds = full_ds.shuffle(dataset_size, seed=seed)

    # Calculate the number of images in each set
    train_size = int(0.7 * dataset_size)
    val_size = int(0.15 * dataset_size)
    test_size = dataset_size - train_size - val_size

    # Shuffle and split the dataset
    train_ds = full_ds.take(train_size)
    val_test_ds = full_ds.skip(train_size)
    val_ds = val_test_ds.take(val_size)
    test_ds = val_test_ds.skip(val_size)

    return train_ds, val_ds, test_ds
```

```python
merged_dir_binary = 'archive/BreaKHis_v1/BreaKHis_v1/histology_slides/breast/merged_images_binary/'

# Example usage for binary classification
binary_train_ds, binary_val_ds, binary_test_ds = load_and_split_dataset(
    directory=merged_dir_binary,
    img_size=(224, 224),
    batch_size=3104
)
```

Figure 19: Split BreakHis Dataset to Train, Validation and Test Set

5. Model Development

5.1 Machine Learning Models (Wisconsin Dataset)

Wisconsin Dataset is trained with 8 different conventional machine learning models and chose the best performing one.

Algorithms Used:

1. Logistic Regression

```
# Define the model
log_reg = LogisticRegression()

# Create StratifiedKFold object for k-fold cross-validation
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
```

```
# Hyperparameter grid for tuning
param_grid = {
    'penalty': ['l2'],
    'C': [0.01, 0.1, 1, 10],
    'solver': ['liblinear', 'lbfgs']
}

# Scorers for metrics
scorers = {
    'accuracy': make_scorer(accuracy_score),
    'precision': make_scorer(precision_score),
    'recall': make_scorer(recall_score),
    'f1': make_scorer(f1_score),
    'roc_auc': make_scorer(roc_auc_score)
}
```

```
# Grid Search with Cross-Validation
grid_search = GridSearchCV(log_reg, param_grid, scoring='accuracy', cv=cv, return_train_score=True)
grid_search.fit(X_train_scaled, y_train)

# Results from Grid Search
print("Best parameters:", grid_search.best_params_)
print("Best score (Accuracy):", grid_search.best_score_)
```

```
Best parameters: {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}
Best score (Accuracy): 0.9736572890025575
```

```
# Calculate metrics for the best estimator
y_pred_eval = grid_search.predict(X_eval_scaled)
accuracy_log = accuracy_score(y_eval, y_pred_eval)
precision_log = precision_score(y_eval, y_pred_eval)
recall_log = recall_score(y_eval, y_pred_eval)
f1_log = f1_score(y_eval, y_pred_eval)
roc_auc_log = roc_auc_score(y_eval, y_pred_eval)

print(f"\nEvaluation Metrics for Logistic Regression:\nAccuracy: {accuracy_log:.4f}\nPrecision: {precision_log:.4f}\nRecall: {recall_log:.4f}\nF1-Score:
```

Figure 20: Model Training – Logistic Regression

2. Random Forest

```
# Define the model
rf = RandomForestClassifier(random_state=42)
```

```
# Hyperparameter grid for tuning
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10]
}

# Grid Search with Cross-Validation
grid_search_rf = GridSearchCV(rf, param_grid, scoring='accuracy', cv=cv, return_train_score=True)
grid_search_rf.fit(X_train, y_train)

# Results from Grid Search
print("Best parameters:", grid_search_rf.best_params_)
print("Best score (Accuracy):", grid_search_rf.best_score_)
```

```
Best parameters: {'max_depth': None, 'min_samples_split': 2, 'n_estimators': 50}
Best score (Accuracy): 0.9531969309462915
```

```
# Calculate metrics for the best estimator
y_pred_eval_rf = grid_search_rf.predict(X_eval)
accuracy_rf = accuracy_score(y_eval, y_pred_eval_rf)
precision_rf = precision_score(y_eval, y_pred_eval_rf)
recall_rf = recall_score(y_eval, y_pred_eval_rf)
f1_rf = f1_score(y_eval, y_pred_eval_rf)
roc_auc_rf = roc_auc_score(y_eval, y_pred_eval_rf)

print(f"\nEvaluation Metrics for Random Forest:\nAccuracy: {accuracy_rf:.4f}\nPrecision: {precision_rf:.4f}\nRecall: {recall_rf:.4f}\nF1-Score: {f1_rf:.4
```

Figure 21: Model Training – Random Forest

3. XGBoost

```python
# Define the model
xgb = XGBClassifier(random_state=42, eval_metric='logloss')
```

```python
# Hyperparameter grid for tuning
param_dist = {
    'n_estimators': [100, 200, 300],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.1, 0.2],
    'subsample': [0.6, 0.8, 1.0]
}

# Random Search with Cross-Validation
# grid_search = GridSearchCV(log_reg, param_grid, scoring='accuracy', cv=cv, return_train_score=True)
random_search_xgb = RandomizedSearchCV(xgb, param_dist, scoring='accuracy', cv=cv, return_train_score=True)
random_search_xgb.fit(X_train, y_train)

# Results from Random Search
print("Best parameters:", random_search_xgb.best_params_)
print("Best score (Accuracy):", random_search_xgb.best_score_)
```

```
Best parameters: {'subsample': 1.0, 'n_estimators': 200, 'max_depth': 7, 'learning_rate': 0.2}
Best score (Accuracy): 0.9766410912190964
```

```python
# Calculate metrics for the best estimator
y_pred_eval = random_search_xgb.predict(X_eval)
accuracy_xgb = accuracy_score(y_eval, y_pred_eval)
precision_xgb = precision_score(y_eval, y_pred_eval)
recall_xgb = recall_score(y_eval, y_pred_eval)
f1_xgb = f1_score(y_eval, y_pred_eval)
roc_auc_xgb = roc_auc_score(y_eval, y_pred_eval)
```

Figure 22: Model Training - XGBoost

4. Gradient Boosting

```python
# Define the model
gb = GradientBoostingClassifier(random_state=42)
```

```python
# Hyperparameter grid for tuning
param_grid_gb = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7]
}

# Grid Search with Cross-Validation
grid_search_gb = GridSearchCV(gb, param_grid_gb, scoring='accuracy', cv=cv, return_train_score=True)
grid_search_gb.fit(X_train, y_train)

# Results from Grid Search
print("Best parameters:", grid_search_gb.best_params_)
print("Best score (Accuracy):", grid_search_gb.best_score_)
```

```
Best parameters: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 300}
Best score (Accuracy): 0.9736572890025575
```

```python
# Calculate metrics for the best estimator
y_pred_eval_gb = grid_search_gb.predict(X_eval)
accuracy_gb = accuracy_score(y_eval, y_pred_eval_gb)
precision_gb = precision_score(y_eval, y_pred_eval_gb)
recall_gb = recall_score(y_eval, y_pred_eval_gb)
f1_gb = f1_score(y_eval, y_pred_eval_gb)
roc_auc_gb = roc_auc_score(y_eval, y_pred_eval_gb)
```

Figure 23: Model Training – Gradient Boosting

5. K-Nearest Neighbours

```python
from sklearn.neighbors import KNeighborsClassifier

# Define the model
knn = KNeighborsClassifier()
```

```python
# Hyperparameter grid for tuning
param_grid_knn = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan']
}

# Grid Search with Cross-Validation
grid_search_knn = GridSearchCV(knn, param_grid_knn, scoring='accuracy', cv=cv, return_train_score=True, verbose=1)
grid_search_knn.fit(X_train, y_train)

# Results from Grid Search
print("Best parameters:", grid_search_knn.best_params_)
print("Best score (Accuracy):", grid_search_knn.best_score_)
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Best parameters: {'metric': 'manhattan', 'n_neighbors': 3, 'weights': 'uniform'}
Best score (Accuracy): 0.9296675191815857
```

```python
# Calculate metrics for the best estimator
y_pred_eval_knn = grid_search_knn.predict(X_eval)
accuracy_knn = accuracy_score(y_eval, y_pred_eval_knn)
precision_knn = precision_score(y_eval, y_pred_eval_knn)
recall_knn = recall_score(y_eval, y_pred_eval_knn)
f1_knn = f1_score(y_eval, y_pred_eval_knn)
roc_auc_knn = roc_auc_score(y_eval, y_pred_eval_knn)
```

Figure 24: Model Training – K-Nearest Neighbours

6. Naïve Bayes

```python
# Define the model
nb_model = GaussianNB()

# Train the model with the entire training set
nb_model.fit(X_train, y_train)

# Make predictions on evaluation data
y_pred_eval_nb = nb_model.predict(X_eval)

# Calculate metrics for evaluation
accuracy_nb = accuracy_score(y_eval, y_pred_eval_nb)
precision_nb = precision_score(y_eval, y_pred_eval_nb)
recall_nb = recall_score(y_eval, y_pred_eval_nb)
f1_nb = f1_score(y_eval, y_pred_eval_nb)
roc_auc_nb = roc_auc_score(y_eval, y_pred_eval_nb)
```

Figure 25: Model Training – Naïve Bayes

7. Stochastic Gradient Decent

```python
# Define the model
sgd_model = SGDClassifier(random_state=42)
```

```python
# Hyperparameter tuning for SGD model
param_grid_sgd = {
    # Loss function to be minimized
        # 1. hinge: linear SVM loss function
        # 2. log_loss: logistic regression loss function
        # 3. modified_huber: Smooth combination of both hinge and log_loss which is less sensitive to outliers
        # 4. square_hinge: loss function which is more sensitive to outliers
    'loss': ['hinge', 'log_loss', 'modified_huber', 'squared_hinge', 'perceptron'],
    # Penalises model complexity reducing overfitting
        # 1. 'l1': L1 regularization (Lasso), leads to sparse solutions (some weights are zero).
        # '2. l2': L2 regularization (Ridge), penalizes large weights, favoring smaller weights.
        # 3. 'elasticnet': Combination of L1 and L2 regularization, balancing between the two.
    'penalty': ['l2', 'l1', 'elasticnet'],
    'alpha': [0.0001, 0.001, 0.01], # Regularization parameter
    'learning_rate': ['constant', 'optimal', 'invscaling', 'adaptive'], # how model adjusts its parameters in each iterations
    'early_stopping': [True],  # Enable early stopping
    'n_iter_no_change': [5, 10],  # Number of iterations with no improvement for early stopping
    'eta0': [0.01, 0.001, 0.0001],  # Initial learning rate (>0)
    'tol': [1e-3, 1e-4] # Model halts training if improvement in loss is smaller than tolerance.
}
```

```python
# Grid Search with Cross-Validation
grid_search_sgd = GridSearchCV(sgd_model, param_grid_sgd, scoring='accuracy', cv=cv, return_train_score=True)
grid_search_sgd.fit(X_train, y_train)
```

```python
# Train the best model found by GridSearchCV
best_sgd = grid_search_sgd.best_estimator_

# Perform predictions on evaluation data
y_pred_eval_sgd = best_sgd.predict(X_eval)
```

Figure 26: Model Training – Stochastic Gradient Decent

8. Decision Tree

```python
# Define the Decision Tree model
dt_model = DecisionTreeClassifier(random_state=42)
```

```python
# Hyperparameter grid for tuning
param_grid_dt = {
    'criterion': ['gini', 'entropy'],        # Split criterion
    'max_depth': [None, 10, 20, 30],         # Depth of the tree
    'min_samples_split': [2, 10, 20],        # Minimum samples required to split a node
    'min_samples_leaf': [1, 5, 10],          # Minimum samples required in a leaf node
    'max_features': [None, 'sqrt', 'log2']  # Number of features to consider for split
}

# Grid Search with Cross-Validation
grid_search_dt = GridSearchCV(dt_model, param_grid_dt, scoring='accuracy', cv=cv, return_train_score=True)
grid_search_dt.fit(X_train, y_train)

# Results from Grid Search
print("Best parameters:", grid_search_dt.best_params_)
print("Best score (Accuracy):", grid_search_dt.best_score_)
```

```
Best parameters: {'criterion': 'entropy', 'max_depth': None, 'max_features': None, 'min_samples_leaf': 5, 'min_samples_split': 2}
Best score (Accuracy): 0.9707161125319693
```

```python
# Train the best model found by GridSearchCV
best_dt = grid_search_dt.best_estimator_

# Perform predictions on evaluation data
y_pred_eval_dt = best_dt.predict(X_eval)
```

Figure 27: Model Training – Decision Tree

In each of the models, we have used Grid Search hyper parameter tuning and K-fold cross validation.

5.2 Deep Learning Models (BreakHis Dataset)

For the BeakHis Image dataset, we create conventional neural networks from scratch and train these for both binary (benign or malignant) and multiclass (8 subclasses) classification problem.

5.2.1 Binary Classification

CNN Architecture:

```
# For Binaryclass model
def create_cnn_model_binary(input_shape=(224, 224, 3), num_classes=2, l2_lambda=0.001):
    model = models.Sequential([
        layers.Input(shape=input_shape),

        # Convolutional Block 1
        layers.Conv2D(32, (3, 3), activation='relu', padding='same', kernel_regularizer=regularizers.l2(l2_lambda), name='conv1_1'),
        layers.Conv2D(32, (3, 3), activation='relu', padding='same', kernel_regularizer=regularizers.l2(l2_lambda), name='conv1_2'),
        layers.MaxPooling2D((2, 2), name='max_pool1'),
        layers.BatchNormalization(),

        # Convolutional Block 2
        layers.Conv2D(64, (3, 3), activation='relu', padding='same', kernel_regularizer=regularizers.l2(l2_lambda), name='conv2_1'),
        layers.Conv2D(64, (3, 3), activation='relu', padding='same', kernel_regularizer=regularizers.l2(l2_lambda), name='conv2_2'),
        layers.MaxPooling2D((2, 2), name='max_pool2'),
        layers.BatchNormalization(),

        # Convolutional Block 3
        layers.Conv2D(128, (3, 3), activation='relu', padding='same', kernel_regularizer=regularizers.l2(l2_lambda), name='conv3_1'),
        layers.Conv2D(128, (3, 3), activation='relu', padding='same', kernel_regularizer=regularizers.l2(l2_lambda), name='conv3_2'),
        layers.MaxPooling2D((2, 2), name='max_pool3'),
        layers.BatchNormalization(),

        # Flatten and Dense Layers
        layers.Flatten(name='flatten'),
        layers.Dense(256, activation='relu', name='dense1', kernel_regularizer=regularizers.l2(l2_lambda)),
        layers.Dropout(0.5),  # Dropout for regularization

        # Output layer based on the number of classes
        layers.Dense(1, activation='sigmoid', name='output')
    ])

    # Select loss function based on number of classes
    loss = 'binary_crossentropy'

    # Compile the model
    model.compile(
        optimizer=optimizers.Adam(learning_rate=1e-5),
        loss=loss,
        metrics=['accuracy']
    )

    return model
```

Figure 28: CNN Model Architecture for Binary Class Classification

Training Configuration:

- Early stopping and learning rate scheduler.

```
# Callbacks
# Early stopping
early_stopping = callbacks.EarlyStopping(
    monitor='val_loss',
    patience=3,
    restore_best_weights=True
)

# Learning rate scheduler
lr_scheduler = callbacks.ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.3,
    patience=3,
    min_lr=1e-6,
    verbose=1
)
```

Figure 29: Early Stopping and LR Scheduler for Binary and Multiclass Models

- Epochs: 20.
- Model saved as .keras.

```python
# Define the ModelCheckpoint callback to save the model after each epoch
checkpoint_callback = ModelCheckpoint(
    'binary_model_checkpoint.keras',  # Path where the model will be saved
    save_best_only=True,  # Only save the best model based on validation loss
    monitor='val_loss',  # Monitor validation loss for saving the best model
    mode='min',  # 'min' means we are looking for the minimum validation loss
    verbose=1  # Print messages when saving the model
)
```

```python
# Train the binary model
history_binary = binary_model.fit(
    binary_train_ds,
    validation_data=binary_val_ds,
    epochs=20,
    callbacks=[early_stopping, lr_scheduler, checkpoint_callback]
)
```

Figure 30: Train Binary Class Model

Testing and Evaluation:

- Testing the binary model with unseen data.

```python
binar_model_loss, binar_model_accuracy = binary_model.evaluate(binary_test_ds)
print(f"Test Loss: {binar_model_loss:.4f}")
print(f"Test Accuracy: {binar_model_accuracy:.4f}")
```

Figure 31: Testing of Binary model

- Precision-recall curve for threshold optimization.
- Optimum threshold calculation (best F1-score) (Fig 34).
- Confusion matrix and classification report (Fig 35).

5.2.2 Multiclass Classification

CNN Architecture:

.

```python
# For Multiclass model
def create_cnn_model(num_classes, input_shape=(224, 224, 3)):
    model = models.Sequential([
        layers.Input(shape=input_shape),

        # Convolutional Block 1
        layers.Conv2D(64, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', name='conv1_1'),
        layers.Conv2D(64, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', name='conv1_2'),
        layers.MaxPooling2D((2, 2), name='max_pool1'),
        layers.BatchNormalization(name='batch_normalization1'),

        # Convolutional Block 2
        layers.Conv2D(128, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', name='conv2_1'),
        layers.Conv2D(128, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', name='conv2_2'),
        layers.MaxPooling2D((2, 2), name='max_pool2'),
        layers.BatchNormalization(name='batch_normalization2'),

        # Convolutional Block 3
        layers.Conv2D(256, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', name='conv3_1'),
        layers.Conv2D(256, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', name='conv3_2'),
        layers.MaxPooling2D((2, 2), name='max_pool3'),
        layers.BatchNormalization(name='batch_normalization3'),

        # Global Average Pooling Layer
        layers.GlobalAveragePooling2D(name='global_avg_pool'),

        # Dense Layers
        layers.Dense(512, activation='relu', kernel_regularizer=regularizers.l2(0.01), name='dense1'),
        layers.BatchNormalization(name='dense_batch_norm'),
        layers.Dropout(0.3),  # Lower dropout

        # Output layer
        layers.Dense(1 if num_classes == 2 else num_classes,
                     activation='sigmoid' if num_classes == 2 else 'softmax',
                     name='output')
    ])

    # Compile the model
    model.compile(
        optimizer=optimizers.Adam(learning_rate=1e-5, clipnorm=1.0),  # Lower learning rate
        loss='binary_crossentropy' if num_classes == 2 else 'categorical_crossentropy',
        metrics=['accuracy']
    )

    return model
```

```python
multiclass_model = create_cnn_model(num_classes=8)
```

Figure 32: CNN Model Architecture for Multiclass Classification

Training Configuration:

- Early stopping and learning rate scheduler (Fig 29).
- Epochs: 50.

```
history_multiclass = multiclass_model.fit(
    multiclass_train_ds,
    validation_data=multiclass_val_ds,
    epochs=50,
    callbacks=[early_stopping, lr_scheduler]
)
```

Figure 33: Training Multi Class Model

Testing and Evaluation:

- Predicting the multiclass model with test dataset.

```
# Evaluate on test data
test_loss, test_accuracy = multiclass_model.evaluate(multiclass_test_ds)
print(f"Test Loss: {test_loss:.4f}")
print(f"Test Accuracy: {test_accuracy:.4f}")

122/122 ━━━━━━━━━━━━━━━━━━━━ 291s 2s/step - accuracy: 0.9194 - loss: 1.4999
Test Loss: 1.5174
Test Accuracy: 0.9098
```

```
# Get true labels and predictions
y_true = []
y_pred = []

for images, labels in multiclass_test_ds:
    y_true.append(np.argmax(labels.numpy(), axis=1))  # One-hot to class index
    y_pred.append(np.argmax(multiclass_model.predict(images), axis=1))

# Flatten lists
y_true = np.concatenate(y_true)
y_pred = np.concatenate(y_pred)
```

Figure 34: Testing and Predicting Output using Multi Class Model

- Classification report and confusion matrix.

6. Results and Analysis

5.1 Wisconsin Dataset

Model Comparisons:

- A table comparing accuracy, precision, recall, F1-score for all ML models.

```
# Round all metrics to 4 decimal places
metrics_df.iloc[:, 1:] = metrics_df.iloc[:, 1:].round(4)

# Printing the metrices dataframe
metrics_df
```

Figure 35: Model Comparison Table for Wisconsin Data

- Testing the best model of Random Forest using test dataset.

```
# Use the best parameters to create the final model
best_rf = RandomForestClassifier(
    max_depth=None,
    min_samples_split=5,
    n_estimators=50,
    random_state=42
)

# Fit the model on the entire training and evaluation dataset
best_rf.fit(X_train_eval, y_train_eval)

# Make predictions on the test set
y_test_pred = best_rf.predict(X_test)

# Calculate performance metrics
accuracy_test = accuracy_score(y_test, y_test_pred)
precision_test = precision_score(y_test, y_test_pred)
recall_test = recall_score(y_test, y_test_pred)
f1_test = f1_score(y_test, y_test_pred)
roc_auc_test = roc_auc_score(y_test, best_rf.predict_proba(X_test)[:, 1])
```

Figure 36: Random Forest Model Prediction

- Highlighting the Random Forest as the best-performing model.

```
# Plot confusion matrix
cm_display = ConfusionMatrixDisplay.from_predictions(y_test, y_test_pred,
                                                     display_labels=['Benign', 'Malignant'], cmap='Blues', colorbar=False)
plt.title("Confusion Matrix Heatmap")
plt.show()
```

```
# Plot ROC curve
roc_display = RocCurveDisplay.from_estimator(best_rf, X_test, y_test)
plt.title("ROC Curve")
plt.show()
```

```
# Plot Precision-Recall curve
pr_display = PrecisionRecallDisplay.from_estimator(best_rf, X_test, y_test)
plt.title("Precision-Recall Curve")
plt.show()
```

```
# Get feature importances
feature_importances = best_rf.feature_importances_
feature_names = X_train_eval.columns
sorted_idx = np.argsort(feature_importances)[::-1]

# Plot feature importances
plt.figure(figsize=(10, 6))
sns.barplot(x=feature_importances[sorted_idx],
            y=feature_names[sorted_idx])
plt.title("Feature Importances")
plt.xlabel("Importance Score")
plt.ylabel("Features")
plt.show()
```

Figure 37: Results of best performing Random Forest model i) Confusion Matric, ii) ROC Curve iii) Precision-Recall Curve iv) Feature Importance plot

5.2 BreakHis Dataset

Binary Classification Results:

- Training vs. Validation Loss and Accuracy.

```
# Extract metrics from the history object
acc = binary_model.history['accuracy']
val_acc = binary_model.history['val_accuracy']
loss = binary_model.history['loss']
val_loss = binary_model.history['val_loss']

# Create line plots
epochs = range(1, len(acc) + 1)

# Plot Accuracy
plt.figure(figsize=(10, 5))
plt.plot(epochs, acc, 'b', label='Training Accuracy')
plt.plot(epochs, val_acc, 'r', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Plot Loss
plt.figure(figsize=(10, 5))
plt.plot(epochs, loss, 'b', label='Training Loss')
plt.plot(epochs, val_loss, 'r', label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Figure 38: Plotting Training and Validation Accuracy and Loss for Binary Model

- Precision-Recall Curve.

```python
for images, labels in binary_test_ds:
    predictions = binary_model.predict(images)
    y_true = np.concatenate([labels], axis=0) # True labels
```

```python
# Calculate precision, recall, and thresholds
precision, recall, thresholds = precision_recall_curve(y_true, predictions)

# Plot Precision-Recall Curve
plt.figure(figsize=(8, 6))
plt.plot(thresholds, precision[:-1], label='Precision', color='b')
plt.plot(thresholds, recall[:-1], label='Recall', color='g')
plt.xlabel('Threshold')
plt.ylabel('Score')
plt.title('Precision-Recall vs Threshold')
plt.legend()
plt.grid()
plt.show()
```

Figure 39: Precision-Recall Curve against Threshold after Testing and Prediction of Binary Model

- Optimum threshold confusion matrix using the best threshold.

```python
# Initialize variables
best_threshold = 0
best_f1 = 0

# Iterate over thresholds
for t in thresholds:
    y_pred = (predictions >= t).astype(int)
    f1 = f1_score(y_true, y_pred)
    if f1 > best_f1:
        best_f1 = f1
        best_threshold = t

print("Best Threshold:", best_threshold)
print("Best F1-Score:", best_f1)
```

```
Best Threshold: 0.8978374
Best F1-Score: 1.0
```

Figure 40: Optimum Threshold Calculation

```python
y_pred = (predictions > 0.89).astype(int) # Predicted labels

cm = confusion_matrix(y_true, y_pred)
cr = classification_report(y_true, y_pred)
```

```python
# Confusion matrix
cm = confusion_matrix(y_true, y_pred)

plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```

Figure 41: Confusion Matrix

Multiclass Classification Results:

- Training vs. Validation Loss and Accuracy (plot).

```python
# Extract metrics from the history object
acc = history_multiclass.history['accuracy']
val_acc = history_multiclass.history['val_accuracy']
loss = history_multiclass.history['loss']
val_loss = history_multiclass.history['val_loss']

# Create line plots
epochs = range(1, len(acc) + 1)

# Plot Accuracy
plt.figure(figsize=(10, 5))
plt.plot(epochs, acc, 'b', label='Training Accuracy')
plt.plot(epochs, val_acc, 'r', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Plot Loss
plt.figure(figsize=(10, 5))
plt.plot(epochs, loss, 'b', label='Training Loss')
plt.plot(epochs, val_loss, 'r', label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Figure 42: Plotting the training and validation accuracy and loss for Multi Class Model

- Final classification report and confusion matrix for 8 classes.

```python
# Print classification report
class_names = [f"Class {i}" for i in range(8)]
report = classification_report(y_true, y_pred, target_names=class_names)
print(report)
```

Figure 43: Classification Report

```python
# Confusion matrix
cm = confusion_matrix(y_true, y_pred)

plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```

Figure 44: Confusion Matrix

6. Running the code:

The code needs to be run in the same order as it has been provided, to avoid the errors being raised for non-declaration of any variables. It is advised run the code in same order, which mean the first cell should be run first followed by the second one.

7. Instructions for Reproducing Results

- Step-by-step guide to execute the project:
  1. Open the .ipynb file and install dependencies
  2. Load datasets.
  3. Run preprocessing scripts.
  4. Train models.
  5. Evaluate models and generate outputs.