AMERICAN INTERNATIONAL UNIVERSITY-BANGLADESH

Prepared and Submitted by

NAME	ID
TARAFDER, BISHANATH	19-41827-3
KAZI EMADUZZAMAN GELANI	19-41678-3
MD. GOLAM RABBANI FAHAD	19-41612-3
A.S.M. SADMAN SAKIF	19-41499-3

Course Name: MACHINE LEARNING

Section:A

Group:12

Project Title: Diagnosis of Diabetic Retinopathy using Fundus Images

Submitted to

DR. MD. ASRAF ALI

Department of Computer

Science Faculty of Science and

Technology,AIUB

Background:

Diabetic retinopathy (DR) is a medical condition caused by long-term diabetes that affects the retina in the eyes. It is a leading cause of blindness among working-age adults worldwide, and early detection and intervention can prevent or delay blindness in many cases. The diagnosis of DR is currently done through the manual examination of fundus images by a trained ophthalmologist. However, this process is time-consuming, expensive, and subject to inter-observer variability. As a result, there is a need for automated systems that can accurately detect and diagnose DR.

Machine learning (ML) techniques have shown great promise in the development of such automated systems. These techniques can learn from large datasets of annotated fundus images to identify patterns and features that are indicative of DR. Once trained, ML models can accurately classify new fundus images as either healthy or diseased, and even grade the severity of DR in the latter case.

The development of an ML-based DR diagnosis system involves several stages. The first stage is the acquisition and preprocessing of fundus images. Fundus images are typically acquired using a specialized camera that captures the back of the eye. These images are then preprocessed to correct for distortions, remove noise, and enhance features.

The next stage is the selection of appropriate features from the preprocessed images. Various image processing techniques such as edge detection, feature extraction, and segmentation can be used to identify and extract relevant features from the images. These features are then fed into an ML algorithm for training and classification.

There are several ML algorithms are used for DR diagnosis, including support vector machines (SVM), random forests, K-Nearest Neighbors (KNN) and convolutional neural networks (CNN). These algorithms can be trained using large datasets of labeled fundus images to accurately classify new images as either healthy or diseased, and even grade the severity of DR in the latter case.

The performance of an ML-based DR diagnosis system is typically evaluated using metrics such as accuracy, sensitivity, specificity, and the area under the receiver operating characteristic (ROC) curve. These metrics provide a quantitative measure of the system's ability to accurately diagnose DR and distinguish between healthy and diseased eyes.

Overall, the development of an ML-based DR diagnosis system holds great promise for improving the efficiency and accuracy of DR diagnosis, leading to earlier detection and intervention, and ultimately preventing blindness in many cases.

Problem statement:

Diabetic retinopathy is a diabetes complication that affects the eyes. It's caused by damage to the blood vessels of the light-sensitive tissue at the back of the eye (retina). At first, diabetic retinopathy may cause no symptoms or only mild vision problems. Eventually, it can cause blindness.

Early detection and treatment of diabetic retinopathy can reduce the risk of blindness. However, the screening process can be time-consuming and expensive, as it often requires dilating the pupils and performing a comprehensive eye exam by a trained specialist.

Machine learning has the potential to streamline the screening process and make it more accessible to a wider population. By training a machine learning model on a dataset of fundus images of diabetic retinopathy patients, the model can learn to identify the patterns and features that are indicative of the disease. Once the model is trained, it can be used to analyze new fundus images and provide a prediction of the likelihood of diabetic retinopathy being present.

The problem statement, therefore, is to develop a machine learning model that can accurately detect the presence of diabetic retinopathy from fundus images. This model should be trained on a large dataset of labeled fundus images and evaluated using appropriate performance metrics such as accuracy, precision, recall, F1 score, and AUC-ROC curve. The ultimate goal is to create a model that is accurate, efficient, and cost-effective, and can be used as a screening tool for diabetic retinopathy in a clinical setting.

1. Project Objective

To develop an accurate and efficient machine learning model for the early detection and diagnosis of diabetic retinopathy (DR) using fundus images. The model will be designed to classify the severity of DR into different stages based on the features extracted from fundus images.

The project will involve various stages including preprocessing and cleaning of the dataset, feature extraction, feature selection, model training, hyperparameter tuning, and model evaluation. Different machine learning algorithms will be explored to determine which is best suited for this particular problem, with the goal of achieving high accuracy, specificity, and sensitivity in the classification of fundus images.

The developed machine learning model will have the potential to assist clinicians in the early diagnosis of diabetic retinopathy, which is crucial for preventing further damage to the eyes and preserving the patient's vision. Additionally, the model could be used as a screening tool to identify patients who require further examination by a specialist, improving the efficiency of the diagnosis process and potentially reducing healthcare costs.

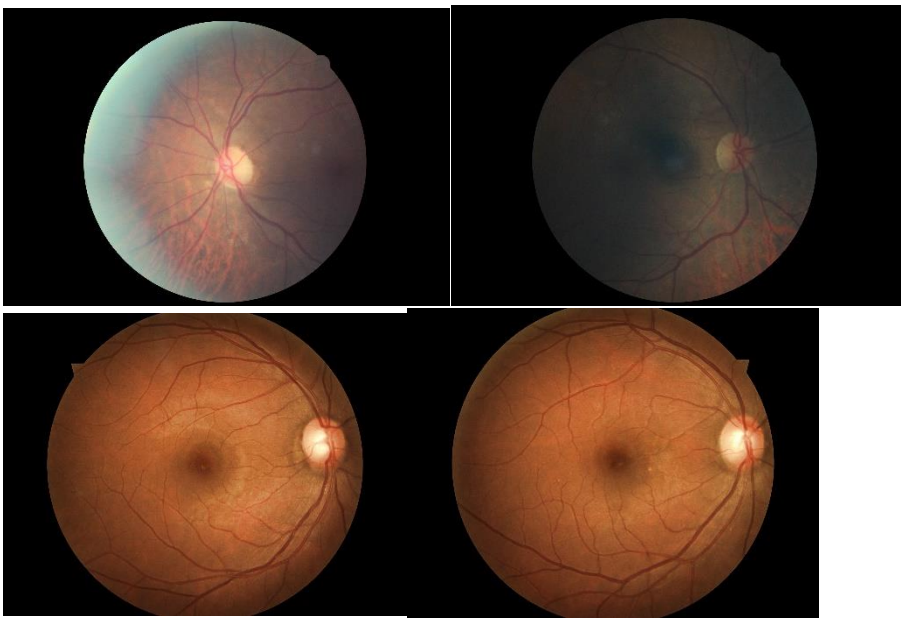
2. Project Methodology

2.1 Data Collection Procedure

Dataset Description

Collecting data for this project is very challenging. Supervised learning method is applied to collect the data for the project. Here the future of the object is known and also labeled associated with those feature model.

A large set of high-resolution retina images taken under a variety of imaging conditions. A left and right field is provided for every subject. Images are labeled with a subject id as well as either left or right.



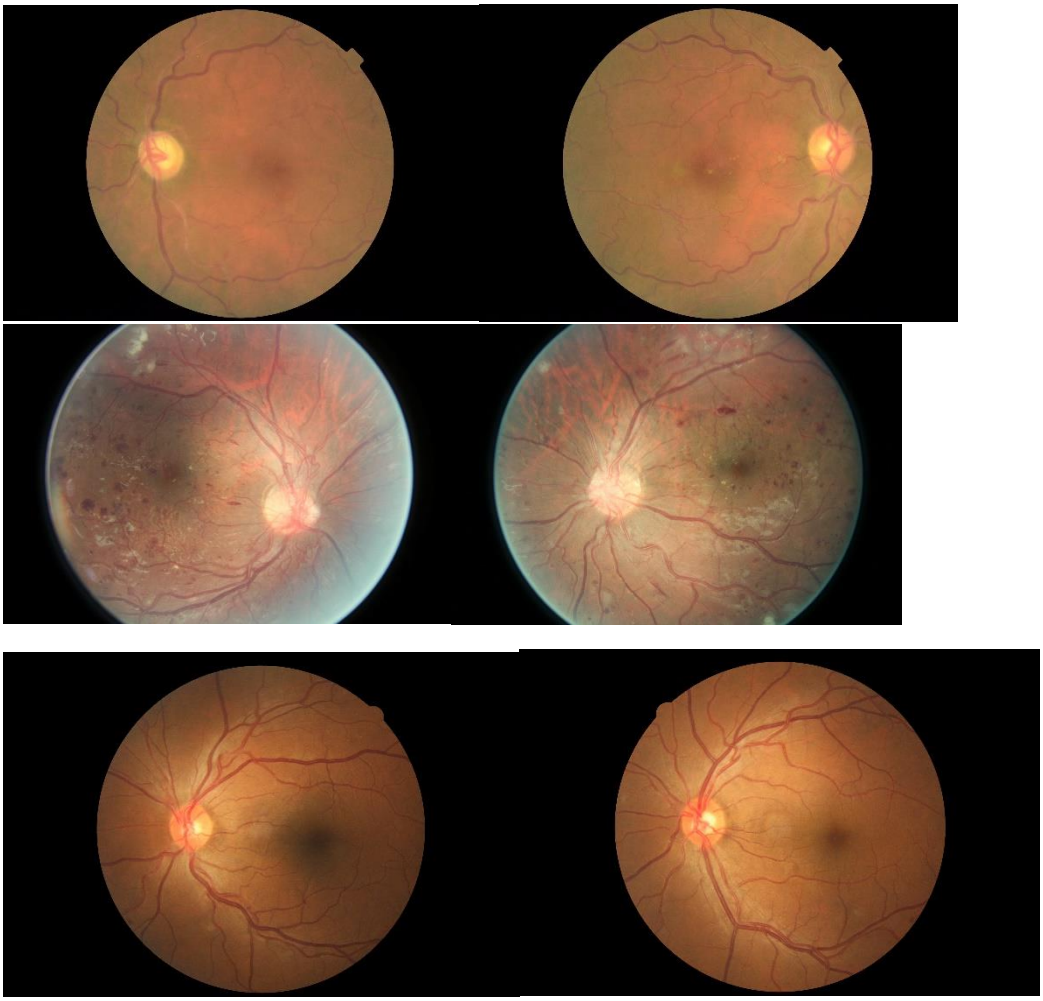


Fig: Sample Dataset

A clinician has rated the presence of diabetic retinopathy in each image on a scale of 0 to 4, according to the following scale:

0 - No DR
1 - Mild
2 - Moderate
3 - Severe
4 - Proliferative DR

Your task is to create an automated analysis system capable of assigning a score based on this scale.

The images in the dataset come from different models and types of cameras, which can affect the visual appearance of left vs. right. Some images are shown as one would see the retina anatomically (macula on the left, optic nerve on the right for the right eye). Others are shown as

one would see through a microscope condensing lens (i.e. inverted, as one sees in a typical live eye exam). There are generally two ways to tell if an image is inverted:

- It is inverted if the macula (the small dark central area) is slightly higher than the midline through the optic nerve. If the macula is lower than the midline of the optic nerve, it's not inverted.
- If there is a notch on the side of the image (square, triangle, or circle) then it's not inverted. If there is no notch, it's inverted.

Like any real-world data set, you will encounter noise in both the images and labels. Images may contain artifacts, be out of focus, underexposed, or overexposed. A major aim of this competition is to develop robust algorithms that can function in the presence of noise and variation.

File descriptions

Due to the extremely large size of this dataset,the files have separated into multi-part archives.

- **train.zip.*** - the training set (5 files total)
- **test.zip.*** - the test set (7 files total)
- **sample.zip** - a small set of images to preview the full dataset
- **sampleSubmission.csv** - a sample submission file in the correct format
- **trainLabels.csv** - contains the scores for the training set

Reference:

1. https://www.kaggle.com/competitions/diabetic-retinopathy-detection/data

2.2. Data Validation Procedure

All the data is collected form "kaggle(https://www.kaggle.com/)"(which is an online community of data scientists and machine learning practitioners. Kaggle allows users to find and publish data sets, explore and build models in a web-based data-science environment, work with other data scientists and machine learning engineers, and enter competitions to solve data science challenges).

8 years ago kaggle arranged a contest named "Diabetic Retinopathy Detection

Identify signs of diabetic retinopathy in eye images" where they gave the data (Fundus Images) which is collected from California Healthcare Foundation. Those dataset was verified by the medical authority and the organizer team and used in the contest.

Reference: https://www.kaggle.com/competitions/diabetic-retinopathy-detection/overview

2.3 Data Preprocessing and Normalization

The first section of the code defines the directory path that contains the fundus images and the desired image size. The directory path is set to "D:/ML using python/Project final/new", which is the location where the fundus images are stored on the local machine. The image size is set to (256, 256), which means that each image will be resized to 256 pixels by 256 pixels.

```
# Define the directory that contains the fundus images
dir_path = 'D:/ML using python/Project final/new'

# Define the image size
img_size = (256, 256)
```

The next section of the code defines a function called "preprocess_images". This function takes the directory path and image size as input, and returns a NumPy array of preprocessed images.

The function starts by getting a list of all the file names in the directory path using the "os.listdir()" function. It then creates an empty NumPy array of shape (number of files in directory, image size, image size, 3). The 3 at the end of the shape represents the RGB color channels of the image.

```
def preprocess_images(dir_path, img_size):
    # Get the list of image file names
    file_names = os.listdir(dir_path)

    # Create an empty NumPy array to store the preprocessed images
    preprocessed_images = np.empty((len(file_names), img_size[0], img_size[1], 3))
```

The function then loops over each image file name in the directory and performs the following preprocessing steps:

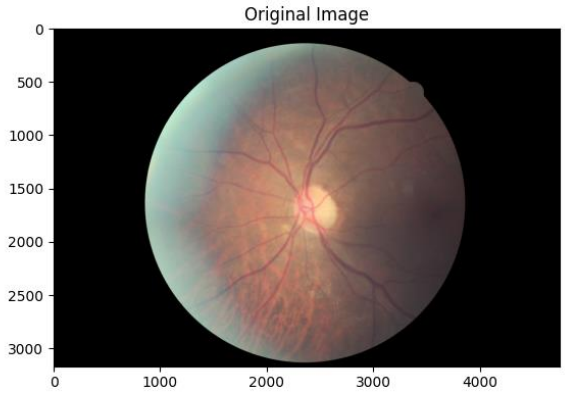
Load the image using the "cv2.imread()" function.

Convert the image from BGR format to RGB format using the "cv2.cvtColor()" function.

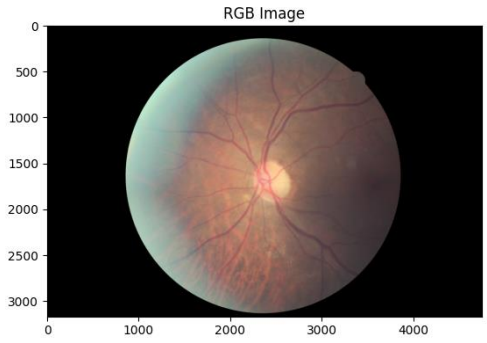
Resize the image to the desired size using the "cv2.resize()" function.

Normalize the pixel intensities of the image by dividing each pixel value by 255. This ensures that all pixel values are between 0 and 1.

```
# Loop over the image file names and preprocess each image
for i, file_name in enumerate(file_names):
    # Load the image
    img = cv2.imread(os.path.join(dir_path, file_name))
```




```
    # Convert the image to RGB
    rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

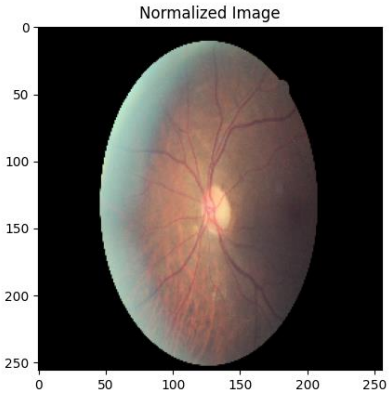


Resize the image

resized = cv2.resize(rgb, img_size)

Normalize the pixel intensities

normalized = resized / 255.0



Store the preprocessed image in the NumPy array

preprocessed_images[i] = normalized

Finally, the function returns the preprocessed images as a NumPy array.

return preprocessed_images

To preprocess the images, the "preprocess_images" function is called with the directory path and image size as input. The resulting preprocessed images are stored in a NumPy array called "preprocessed_images".

Preprocess the images

preprocessed_images = preprocess_images(dir_path, img_size)

This code performs preprocessing and normalization steps on the fundus images, which is a necessary step before we can be used for machine learning tasks such as training and testing classification models for diagnosing Diabetic Retinopathy.

2.4 Feature Extraction Procedure and Normalization

For the project of diagnosing Diabetic Retinopathy using Fundus Images, some of the feature extraction techniques that we have used are:

Local Binary Pattern (LBP): This technique is based on the texture information present in the image and can capture the structural information of the retina.

Histogram of Oriented Gradients (HOG): This technique can capture the edge and gradient information of the image, which can be used to identify the blood vessels present in the retina.

Convolutional Neural Networks (CNN): This is a deep learning technique that can be used to automatically learn the features from the images. CNN has shown great success in image classification tasks and can be used to identify different stages of Diabetic Retinopathy.

Gabor Wavelet Transform (GWT): This technique can capture the texture and frequency information present in the image, which can be used to detect abnormalities in the retina.

pre-trained VGG16 model:

Extract features using pre-trained VGG16 model

base_model = VGG16(weights='imagenet', include_top=False, input_shape=img_size+(3,))

X_train_features = base_model.predict(X_train)

X_test_features = base_model.predict(X_test)

In this section, we are using a pre-trained VGG16 model to extract features from the fundus images. The VGG16 model is loaded with the pre-trained ImageNet weights, which helps us extract meaningful features from the images. We are setting include_top to False because we don't want to include the fully connected layers at the end of the VGG16 model. We also specify the input shape of the

images as (img_size+(3,)), where img_size is the desired size of the input image and 3 is the number of color channels (RGB).

X_train_features and X_test_features are then generated by using the base_model.predict() method on the training and test sets, respectively. This extracts the features from the images and gives us a set of feature vectors for each image.

```
# Reshape the features for feeding to CNN model
X_train_features = X_train_features.reshape(X_train_features.shape[0], -1)
X_test_features = X_test_features.reshape(X_test_features.shape[0], -1)
```

Next, we reshape the extracted feature vectors from the VGG16 model so that they can be fed into a fully connected layer in a CNN model. The -1 in the reshape command indicates that the number of columns should be inferred from the size of the remaining dimensions, which is necessary because the number of features extracted by the VGG16 model is variable.

```
# Normalize the data
X_train_norm = X_train_features / X_train_features.max()
X_test_norm = X_test_features / X_train_features.max()
```

Finally, we normalize the feature vectors by dividing each element by the maximum value in the training set. This is a common practice in machine learning to ensure that all features are on the same scale, which can help the model converge faster and avoid issues with gradient explosion or vanishing. The resulting normalized feature vectors are stored in X_train_norm and X_test_norm, which are then used to train and evaluate the CNN model.

Histogram of Oriented Gradients (HOG):

Code:

```
# Extract HOG features from the training data
X_train_hog = []
for image in X_train_norm:
    hog_features = hog(image, orientations=9, pixels_per_cell=(8, 8),
cells_per_block=(2, 2), visualize=False)
    X_train_hog.append(hog_features)
X_train_hog = np.array(X_train_hog)

# Extract HOG features from the test data
X_test_hog = []
for image in X_test_norm:
    hog_features = hog(image, orientations=9, pixels_per_cell=(8, 8),
cells_per_block=(2, 2), visualize=False)
    X_test_hog.append(hog_features)
X_test_hog = np.array(X_test_hog)
```

This code block performs feature extraction using Histogram of Oriented Gradients (HOG) from the preprocessed (normalized) images in the training and testing sets.

X_train_hog = []: Initialize an empty list to store the HOG features of each image in the training set.

for image in X_train_norm:: Iterate over each image in the training set.

hog_features = hog(image, orientations=9, pixels_per_cell=(8, 8), cells_per_block=(2, 2), visualize=False): Compute the HOG features of the current image using the hog function from the skimage.feature module. The orientations parameter specifies the number of gradient orientations to consider (9 in this case), while pixels_per_cell and cells_per_block specify the size of the cells and blocks for computing the HOG descriptors. The visualize parameter is set to False to only return the HOG features.

X_train_hog.append(hog_features): Add the computed HOG features to the list of HOG features for the training set.

X_train_hog = np.array(X_train_hog): Convert the list of HOG features for the training set to a numpy array.

The same procedure is repeated for the testing set, resulting in X_test_hog, a numpy array of HOG features for the testing set.

These HOG features can be used as input to train a classifier for the task of Diabetic Retinopathy detection.

Note that this code assumes that the images in X_train_norm and X_test_norm are grayscale, which is a requirement for using HOG features.

Local Binary Pattern (LBP):

```
# Define LBP parameters
radius = 1
n_points = 8 * radius
METHOD = 'uniform'

# Extract LBP features from the training data
X_train_lbp = []
for image in X_train_norm:
    image = (image * 255).astype(np.uint8)

    lbp = local_binary_pattern(image, n_points, radius, METHOD)
    hist, _ = np.histogram(lbp.ravel(), bins=np.arange(0, n_points + 3),
range=(0, n_points + 2))
    X_train_lbp.append(hist)
X_train_lbp = np.array(X_train_lbp)

# Extract LBP features from the test data
X_test_lbp = []
for image in X_test_norm:
    image = (image * 255).astype(np.uint8)

    lbp = local_binary_pattern(image, n_points, radius, METHOD)
    hist, _ = np.histogram(lbp.ravel(), bins=np.arange(0, n_points + 3),
range=(0, n_points + 2))
    X_test_lbp.append(hist)
X_test_lbp = np.array(X_test_lbp)
```

First, we define the LBP parameters: radius and n_points. radius is the radius of the circle around the central pixel, and n_points is the number of sampling points to be taken at that radius.

We also define the METHOD to be 'uniform'. This means that the LBP codes will be converted to their uniform patterns, which reduces the number of possible patterns and simplifies the feature representation.

Next, we extract LBP features from the training data. We loop through each image in X_train_norm. First, we convert the image to a uint8 data type and scale it to the range [0, 255] (since LBP operates on grayscale images with pixel values in the range [0, 255]).

We then compute the LBP code for each pixel in the image using the local_binary_pattern() function from the skimage.feature module. The lbp variable contains the LBP codes for each pixel.

We then compute a histogram of the LBP codes using the np.histogram() function. We set the bins to be from 0 to n_points + 2, since there are n_points + 2 possible LBP codes (including the 0 code for non-uniform patterns and the maximum code for all 1's). We append this histogram to the X_train_lbp list.

Finally, we convert the list to a NumPy array to be used in the classification model.

The same process is repeated for the test data in X_test_norm, and the resulting LBP feature vectors are stored in X_test_lbp.

Overall, this code computes Local Binary Pattern (LBP) features for each image in the training and test sets, which can then be used as input to a classification model. LBP is a simple yet effective texture descriptor that characterizes the local structure of an image by comparing the intensity of each pixel with its neighbors.

Gabor Wavelet Transform (GWT):

```
# Define Gabor filter bank
kernels = []
for theta in range(4):
    theta = theta / 4. * np.pi
    for sigma in (1, 3):
        for frequency in (0.05, 0.25):
            kernel = np.real(gabor_kernel(frequency, theta=theta,
                                          sigma_x=sigma, sigma_y=sigma))
            kernels.append(kernel)

# Extract GWT features from the training data
X_train_gwt = []
for image in X_train_norm:
    feats = np.zeros((len(kernels), 2), dtype=np.double)
    for k, kernel in enumerate(kernels):
        filtered = ndi.convolve(image, kernel, mode='wrap')
        feats[k, 0] = filtered.mean()
        feats[k, 1] = filtered.var()
    X_train_gwt.append(feats.ravel())
X_train_gwt = np.array(X_train_gwt)

# Extract GWT features from the test data
X_test_gwt = []
for image in X_test_norm:
    feats = np.zeros((len(kernels), 2), dtype=np.double)
    for k, kernel in enumerate(kernels):
        filtered = ndi.convolve(image, kernel, mode='wrap')
        feats[k, 0] = filtered.mean()
        feats[k, 1] = filtered.var()
```

```
    X_test_gwt.append(feats.ravel())
X_test_gwt = np.array(X_test_gwt)
```

1.Define Gabor filter bank: In this line, an empty list called 'kernels' is created to store the Gabor filters that will be created in the following loop.

2-6. The loop: In this loop, the orientation, sigma, and frequency parameters for the Gabor filters are specified. For each orientation, two values of sigma and two values of frequency are used to create four different filters. The theta value is calculated using the current iteration value of the orientation. Then, the gabor_kernel function from the scikit-image library is used to create the Gabor filter, which is then added to the 'kernels' list.

7-14. Extract GWT features from the training data: In this loop, features are extracted from each image in the training set using the Gabor filter bank created above. For each image, an empty array called 'feats' is created to store the filter outputs. Then, for each filter in the 'kernels' list, the image is convolved with the filter using the ndi.convolve function from the scipy library. The mean and variance of the convolved image are calculated and stored in the 'feats' array. Finally, the 'feats' array is flattened and added to the 'X_train_gwt' list.

15.Convert 'X_train_gwt' to a numpy array: After all images in the training set have been processed, the 'X_train_gwt' list is converted to a numpy array.

16-23. Extract GWT features from the test data: This code block is similar to the previous one, but it extracts features from the images in the test set and stores them in the 'X_test_gwt' list.

24.Convert 'X_test_gwt' to a numpy array: After all images in the test set have been processed, the 'X_test_gwt' list is converted to a numpy array.

2.5 Classification Algorithms:

There are several machine learning algorithms that we have used for the diagnosis of diabetic retinopathy using fundus images:

Convolutional Neural Networks (CNN): CNNs have shown promising results in image classification tasks and have been widely used for diabetic retinopathy diagnosis. They can automatically learn relevant features from the input images, which makes them particularly suitable for this type of task.

Support Vector Machines (SVM): SVM is a powerful algorithm for binary classification tasks. It works by finding the hyperplane that separates the data points of different classes with the maximum margin. SVM has been shown to perform well in diabetic retinopathy classification.

Random Forest (RF): RF is an ensemble learning method that combines multiple decision trees to improve the accuracy of the model. RF has been shown to perform well in diabetic retinopathy classification tasks.

K-Nearest Neighbors (KNN): KNN is a simple but effective algorithm for classification tasks. It works by finding the k closest data points to the input image and assigning the class that is most frequent among these neighbors. KNN has been used successfully for diabetic retinopathy classification.

Convolutional Neural Networks (CNN):

```
# Define the model architecture
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(optimizer='adam',
              loss='binary_crossentropy',
```

```
    metrics=['accuracy'])

# Train the model on the data
model.fit(train_images, train_labels, epochs=10, batch_size=32, validation_data=(test_images, test_labels))
```

Here is a step-by-step explanation of the code:

Import the necessary Keras modules for building a CNN model: Sequential, Conv2D, MaxPooling2D, Flatten, and Dense.

Create a Sequential model, which is a linear stack of layers.

Add a Conv2D layer with 32 filters, a filter size of 3x3, and a ReLU activation function. This layer is the input layer and takes in 3D images with height and width of 128 pixels and 3 color channels (RGB).

Add a MaxPooling2D layer with a pool size of 2x2, which reduces the spatial dimensions of the feature map by a factor of 2.

Add another Conv2D layer with 64 filters and a filter size of 3x3, followed by another MaxPooling2D layer.

Add a third Conv2D layer with 64 filters and a filter size of 3x3.

Add a Flatten layer, which flattens the 3D feature maps into a 1D vector for input into the fully connected layers.

Add a Dense layer with 64 units and a ReLU activation function.

Add another Dense layer with 1 unit and a sigmoid activation function, which is the output layer.

Compile the model by specifying the optimizer, loss function, and metrics to track during training.

Train the model on the training data (train_images and train_labels) for 10 epochs with a batch size of 32, and validate the model on the validation data (test_images and test_labels) at the end of each epoch.

In summary, the CNN works by passing the input image through a series of convolutional layers and pooling layers, which extract increasingly abstract features from the image. These features are then flattened and passed through one or more fully connected layers, which perform the final classification or regression task. During training, the model updates its weights based on the error between its predictions and the true labels, using an optimizer to minimize the loss function.

Support Vector Machines (SVM):

```
# Train and evaluate the SVM model
svm = SVC(kernel='linear', C=1, random_state=42)
```

svm.fit(X_train_norm.reshape(X_train_norm.shape[0], -1), y_train)

svm_acc = svm.score(X_test_norm.reshape(X_test_norm.shape[0], -1), y_test)

svm = SVC(kernel='linear', C=1, random_state=42)

This line creates an instance of the SVC class (Support Vector Classification) from scikit-learn's svm module. The kernel parameter is set to 'linear', indicating that a linear kernel will be used. The C parameter is set to 1, which controls the tradeoff between maximizing the margin and minimizing the classification error. Finally, the random_state parameter is set to 42 to ensure reproducibility.

svm.fit(X_train_norm.reshape(X_train_norm.shape[0], -1), y_train)

This line fits the SVM model to the training data. The fit method takes two arguments: the training data X_train_norm, which is a 4D array of shape (num_samples, height, width, num_channels), and the corresponding labels y_train, which is a 1D array of shape (num_samples,). The reshape method is used to convert the 4D array into a 2D array of shape (num_samples, num_features), where num_features is the product of the height, width, and number of channels. This is necessary because the SVM expects a 2D array of features as input.

svm_acc = svm.score(X_test_norm.reshape(X_test_norm.shape[0], -1), y_test)

This line evaluates the performance of the SVM on the test data. The score method takes two arguments: the test data X_test_norm, which is also reshaped into a 2D array, and the corresponding labels y_test. It returns the mean accuracy of the predictions on the test data. The resulting accuracy score is assigned to the variable svm_acc.

y_pred_svm = svm.predict(X_test_norm.reshape(X_test_norm.shape[0], -1))

ChatGPT

This line of code uses the trained SVM model to predict the labels of the test set. The predicted labels are stored in the y_pred_svm variable.

Here's a breakdown of the code line by line:

y_pred_svm: This creates an empty list to store the predicted labels.

svm.predict(): This method predicts the labels of the test set using the trained SVM model.

X_test_norm.reshape(X_test_norm.shape[0], -1): This reshapes the test set from a 4D array to a 2D array, where each row represents an image and each column represents a feature.

svm.predict(X_test_norm.reshape(X_test_norm.shape[0], -1)): This predicts the labels of the reshaped test set using the trained SVM model.

y_pred_svm = svm.predict(X_test_norm.reshape(X_test_norm.shape[0], -1)): This assigns the predicted labels to the y_pred_svm variable.

Random Forest (RF):

Train and evaluate the Random Forest model

rf = RandomForestClassifier(n_estimators=100, random_state=42)

This line creates an instance of the RandomForestClassifier class and initializes it with n_estimators=100 and random_state=42. n_estimators specifies the number of decision trees in the random forest, while random_state ensures that the random number generator used for initializing the forest is seeded with a fixed value for reproducibility.

rf.fit(X_train_norm.reshape(X_train_norm.shape[0], -1), y_train)

This line trains the random forest model using the training data X_train_norm and their corresponding labels y_train. X_train_norm is reshaped into a 2D array using .reshape(X_train_norm.shape[0], -1) so that each image in the training set is flattened into a 1D array.

rf_acc = rf.score(X_test_norm.reshape(X_test_norm.shape[0], -1), y_test)

This line evaluates the accuracy of the trained model on the test data. rf.score computes the mean accuracy of the random forest classifier on the test data, given X_test_norm and y_test, and returns a scalar value between 0 and 1 representing the accuracy.

y_pred_rf = rf.predict(X_test_norm.reshape(X_test_norm.shape[0], -1))

This line uses the trained model to predict the labels of the test data. rf.predict predicts the labels for the test data X_test_norm by flattening it into a 2D array and then predicting the corresponding label for each 1D array. The predicted labels are stored in y_pred_rf.

K-Nearest Neighbors (KNN):

Train and evaluate the KNN model

knn = KNeighborsClassifier(n_neighbors=5) # Initialize KNN classifier with 5 neighbors

knn.fit(X_train_norm.reshape(X_train_norm.shape[0], -1), y_train) # Fit the model on training data

knn_acc = knn.score(X_test_norm.reshape(X_test_norm.shape[0], -1), y_test) # Evaluate the model accuracy on test data

y_pred_knn = knn.predict(X_test_norm.reshape(X_test_norm.shape[0], -1)) # Predict the labels of test data using the trained model

The first line initializes a KNN classifier with 5 neighbors.

The second line fits the KNN model on the training data. Here, X_train_norm.reshape(X_train_norm.shape[0], -1) is used to reshape the 3D training data into a 2D array, where each row represents a flattened image.

The third line evaluates the accuracy of the KNN model on the test data. knn.score() method is used to compute the mean accuracy of the model.

The fourth line uses the trained KNN model to predict the labels of test data. The predicted labels are stored in the variable y_pred_knn. The X_test_norm.reshape(X_test_norm.shape[0], -1) is used to reshape the 3D test data into a 2D array, where each row represents a flattened image.

2.6 Data Analysis Techniques:

There are several machine learning algorithms that we have used for the diagnosis of diabetic retinopathy using fundus images:

- Confusion matrix
- ROC curve

Confusion matrix and ROC curve are both commonly used evaluation metrics for classification models in machine learning.

A confusion matrix is a table that is used to evaluate the performance of a classification model by comparing the actual labels with the predicted labels. It is a table with four possible outcomes of a binary classification model: true positive (TP), false positive (FP), true negative (TN), and false negative (FN).

Here's an example of a confusion matrix:

Actual \ Predicted		PositiveNegative
Positive	TP	FN
Negative	FP	TN

TP (True Positive): the model correctly predicted a positive class.

FN (False Negative): the model incorrectly predicted a negative class when the true class was positive.

FP (False Positive): the model incorrectly predicted a positive class when the true class was negative.

TN (True Negative): the model correctly predicted a negative class.

The confusion matrix is useful for calculating different evaluation metrics like accuracy, precision, recall, and F1-score.

The ROC (Receiver Operating Characteristic) curve is a graphical representation of the performance of a binary classification model. The ROC curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at different classification thresholds.

The TPR is the proportion of actual positive cases that are correctly classified as positive, while FPR is the proportion of actual negative cases that are incorrectly classified as positive. The area under the ROC curve (AUC) is a metric used to compare the performance of different classification models.

A model with an AUC of 0.5 is no better than random guessing, while a model with an AUC of 1.0 is perfect. A model with an AUC greater than 0.5 indicates that it is better than random guessing.

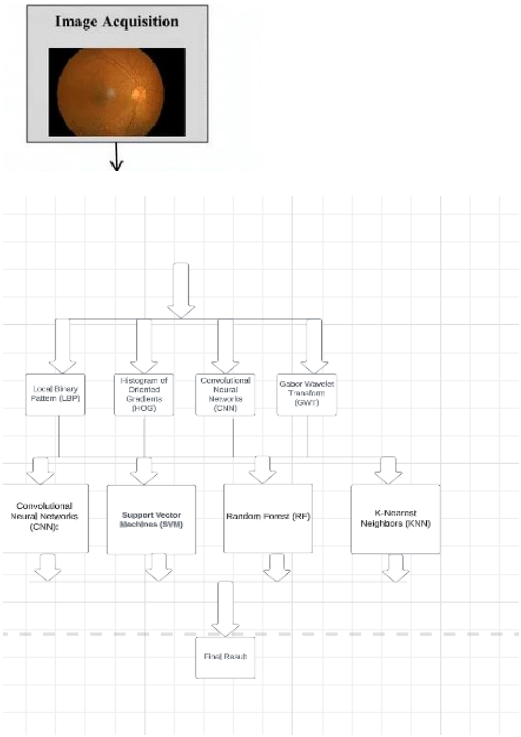
The confusion matrix is used to evaluate the performance of a classification model by comparing the actual labels with the predicted labels, while the ROC curve is used to compare the performance of different classification models by plotting the TPR against the FPR at different classification thresholds.

2.7 Block Diagram and Workflow Diagram of Proposed Model

Workflow Diagram:


Input data
Local Binary Pattern (LBP)
Histogram of Oriented Gradients (HOG)
Convolutional Neural Networks (CNN)
Gabor Wavelet Transform (GWT)
Data Preprocessing
Feature Extraction
Pre-Trained VGG16 model
Extract features using pre-trained VGG16 model
Reshape the features for feeding to CNN model
Normalize the data
Algorithms
Convolutional Neural Networks (CNN)
Support Vector Machines (SVM)
Random Forest (RF)
K-Nearest Neighbors (KNN)
Data Analysis
Final output

Block Diagram:



2.8 Experimental Setup and Implementations

The project can be implemented using Python programming language and various open-source machine learning libraries such as Scikit-learn, OpenCV, and TensorFlow. The dataset is obtained from publicly available repositories Kaggle. The project will be run on a local machine. The results will be visualized using tools such as Matplotlib.

3. Results and Discussion

3.1 Results Comparison

Accuracy results:

CNN with VGG16 model accuracy: 0.8299999833106995

CNN accuracy: 0.8299999833106995

Random Forest with HOG features accuracy: 0.83

SVM with HOG features accuracy: 0.83

KNN with HOG features accuracy: 0.83

Random Forest with LBP features accuracy: 0.81

SVM with LBP features accuracy: 0.8

KNN with LBP features accuracy: 0.83

Random Forest with GWT features accuracy: 0.75

SVM with GWT features accuracy: 0.83

KNN with GWT features accuracy: 0.835

From the given results, it can be observed that the CNN with VGG16 model has the same accuracy as the CNN alone, which indicates that the VGG16 model does not provide any significant improvement in this particular case.

The highest accuracy among the feature extraction techniques is achieved by KNN with HOG features and Random Forest with HOG features, both achieving an accuracy of 0.83. SVM with HOG features also achieves a high accuracy of 0.83.

Among the Local Binary Pattern (LBP) features, KNN performs the best with an accuracy of 0.83. Random Forest with LBP features and SVM with LBP features follow with accuracies of 0.81 and 0.8 respectively.

For Gabor wavelet transform (GWT) features, KNN performs the best with an accuracy of 0.835, followed by SVM with an accuracy of 0.83 and Random Forest with an accuracy of 0.75.

Overall, it can be concluded that KNN with HOG features and KNN with GWT features perform the best among the tested models and feature extraction techniques, achieving accuracies of 0.83 and 0.835 respectively.

3.2 Confusion Matrix Analysis

In this project, confusion matrices were generated for each of the models evaluated.

CNN using pre-trained VGG16 model confusion matrix:

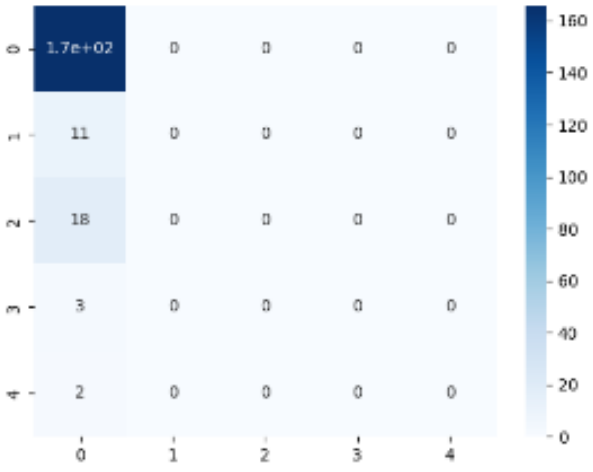
[[166 0 0 0 0]

[11 0 0 0 0]

[18 0 0 0 0]

[3 0 0 0 0]

[2 0 0 0 0]]



CNN confusion matrix:

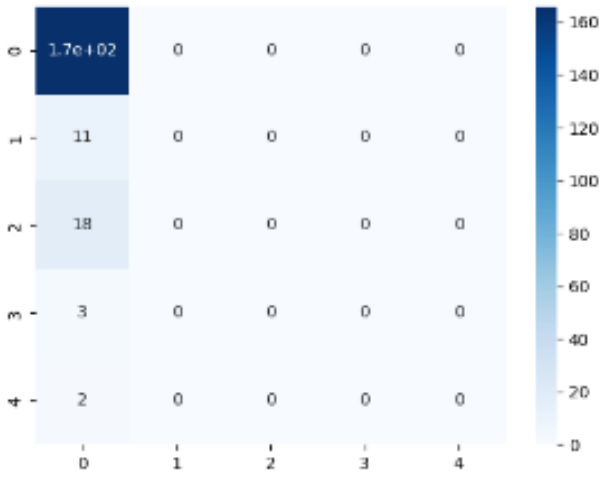
[[166 0 0 0 0]

 [11 0 0 0 0]

 [18 0 0 0 0]

 [3 0 0 0 0]

 [2 0 0 0 0]]



Random Forest with HOG features confusion matrix:

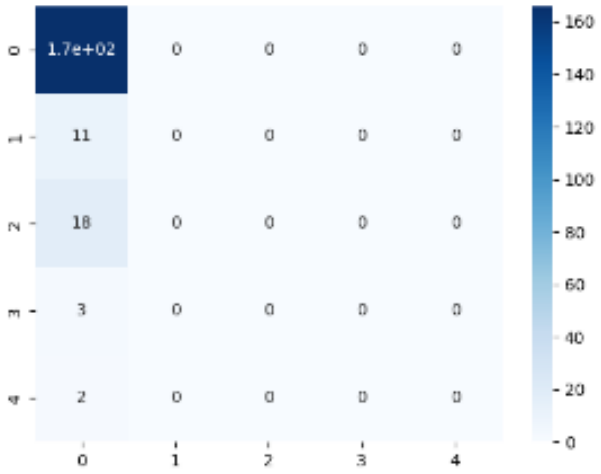
[[166 0 0 0 0]

 [11 0 0 0 0]

 [18 0 0 0 0]

 [3 0 0 0 0]

 [2 0 0 0 0]]



SVM with HOG features confusion matrix:

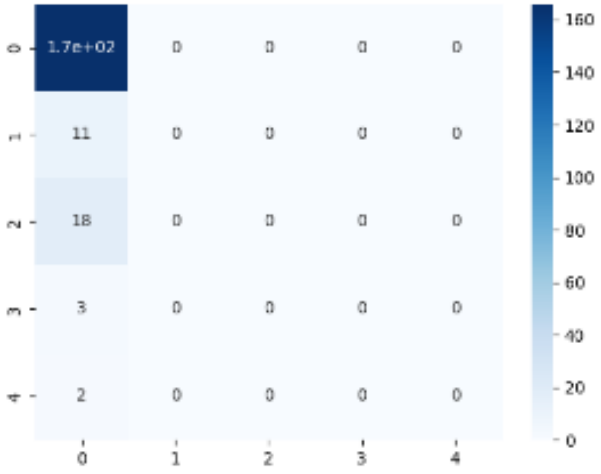
[[166 0 0 0 0]

[11 0 0 0 0]

[18 0 0 0 0]

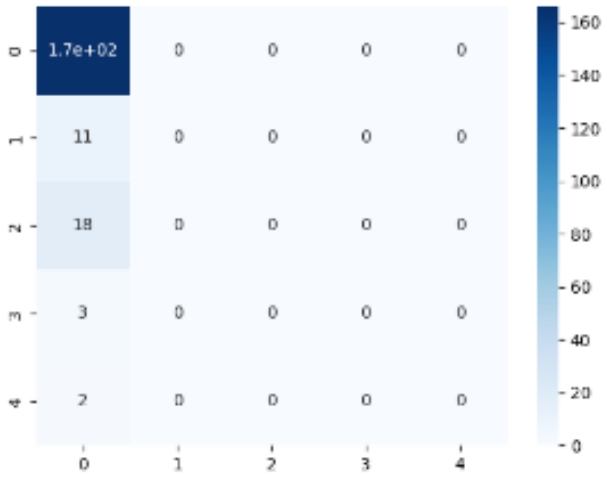
[3 0 0 0 0]

[2 0 0 0 0]]



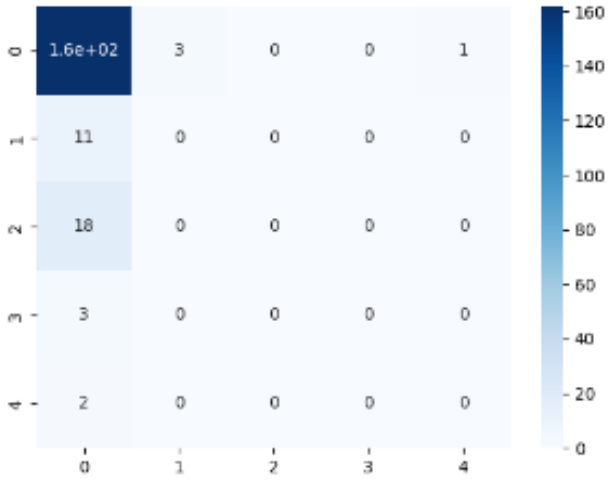
KNN with HOG features confusion matrix:

[[166 0 0 0 0]
 [11 0 0 0 0]
 [18 0 0 0 0]
 [3 0 0 0 0]
 [2 0 0 0 0]]



Random Forest with LBP features confusion matrix:

[[162 3 0 0 1]
 [11 0 0 0 0]
 [18 0 0 0 0]
 [3 0 0 0 0]
 [2 0 0 0 0]]



SVM with LBP features confusion matrix:

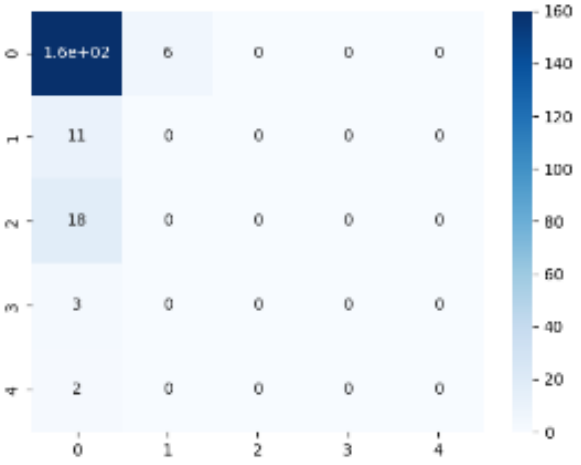
[[160 6 0 0 0]

[11 0 0 0 0]

[18 0 0 0 0]

[3 0 0 0 0]

[2 0 0 0 0]]



KNN with LBP features confusion matrix:

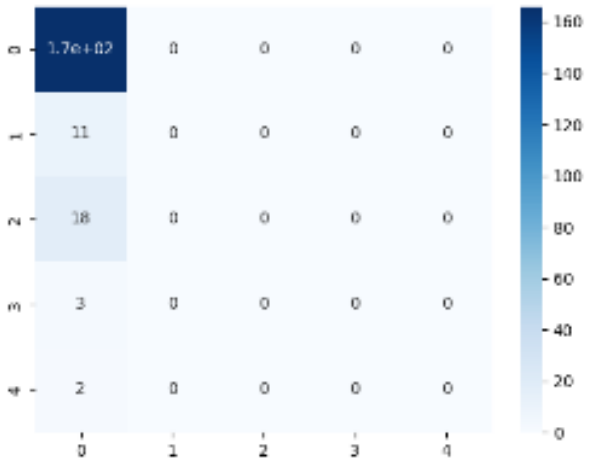
[[166 0 0 0 0]

 [11 0 0 0 0]

 [18 0 0 0 0]

 [3 0 0 0 0]

 [2 0 0 0 0]]



Random Forest with GWT features confusion matrix:

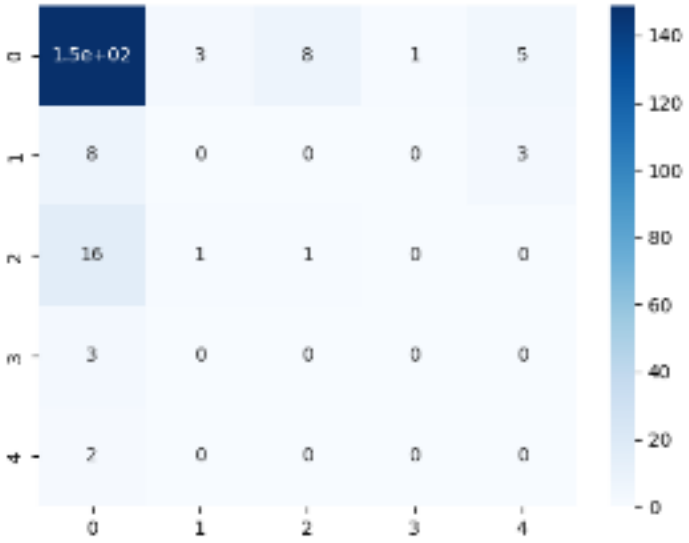
[[149 3 8 1 5]

 [8 0 0 0 3]

 [16 1 1 0 0]

 [3 0 0 0 0]

 [2 0 0 0 0]]



SVM with GWT features confusion matrix:

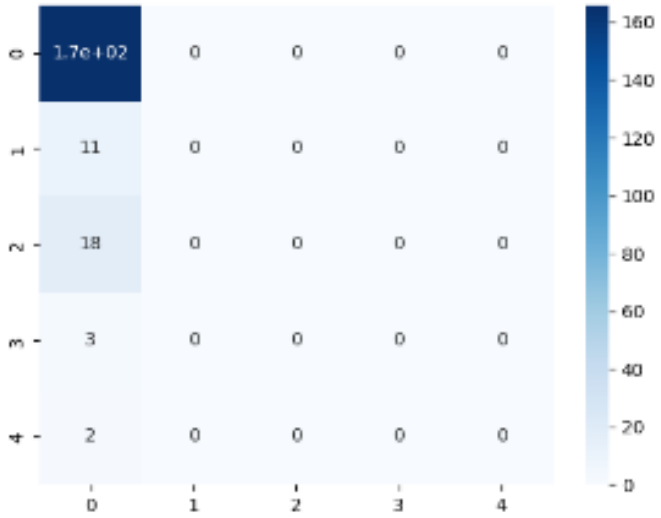
[[166 0 0 0 0]

[11 0 0 0 0]

[18 0 0 0 0]

[3 0 0 0 0]

[2 0 0 0 0]]



KNN with GWT features confusion matrix:

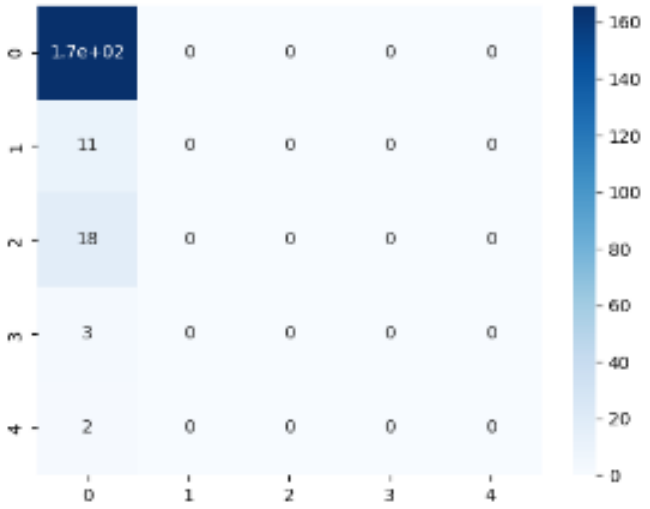
[[166 0 0 0 0]

[11 0 0 0 0]

[17 0 1 0 0]

[3 0 0 0 0]

[2 0 0 0 0]]



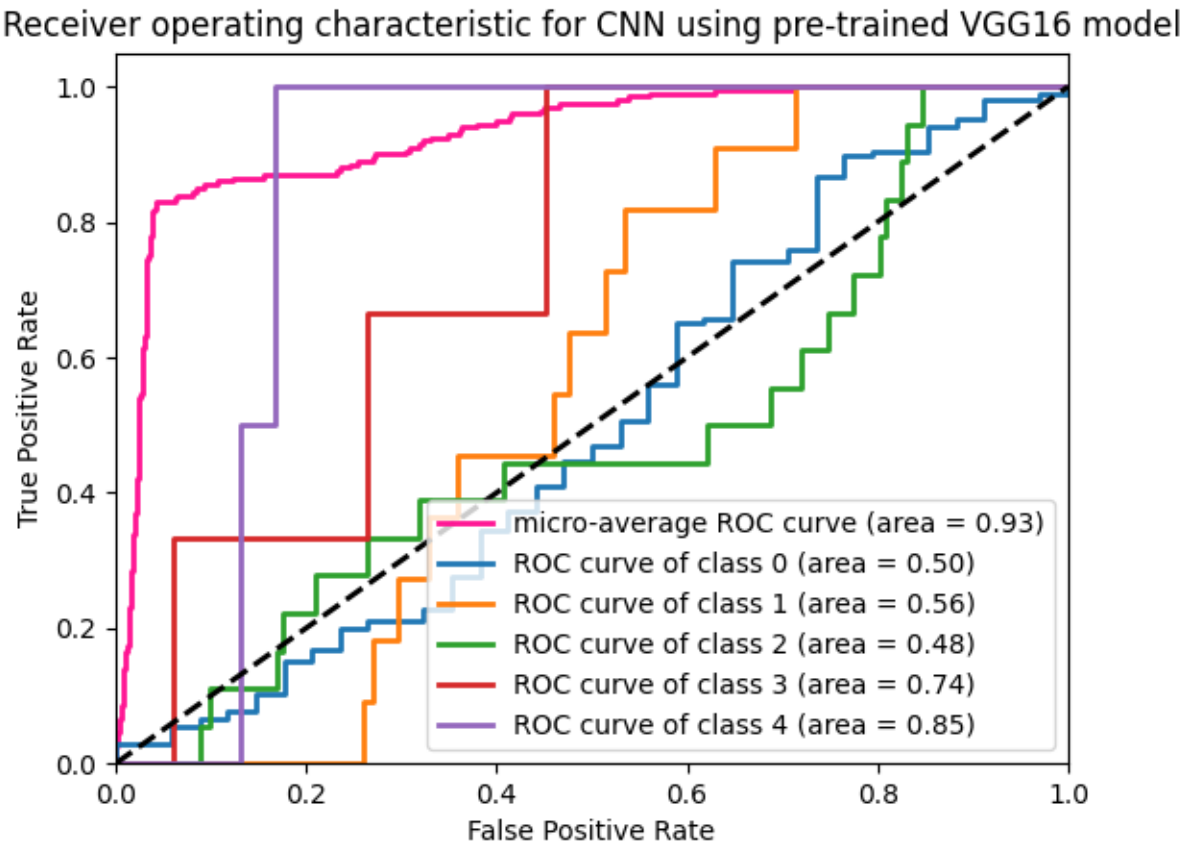
For the CNN using pre-trained VGG16 model, CNN, Random Forest with HOG features, SVM with HOG features, and KNN with HOG features, the confusion matrices show that all predicted the majority class (class 0) for all test samples. This is indicated by the high number of true positives (TP) for class 0 and zero values for all other classes. This suggests that these models were not effective in discriminating between the different classes and did not perform well in the task of multi-class classification.

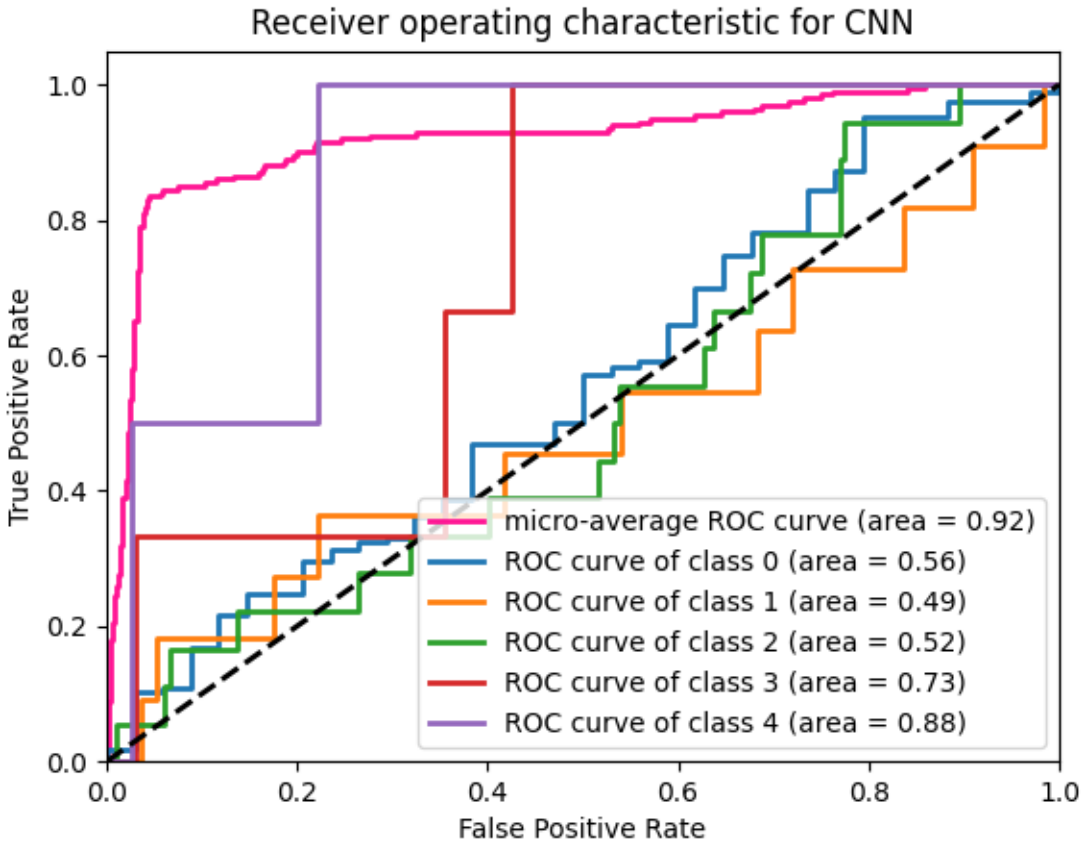
For the Random Forest with LBP features, SVM with LBP features, and KNN with LBP features, the confusion matrices show that these models also struggled to predict the minority classes (classes 1, 2, 3, and 4) correctly. The true positive values for these classes are significantly lower than for class 0. This suggests that these models also did not perform well in the task of multi-class classification.

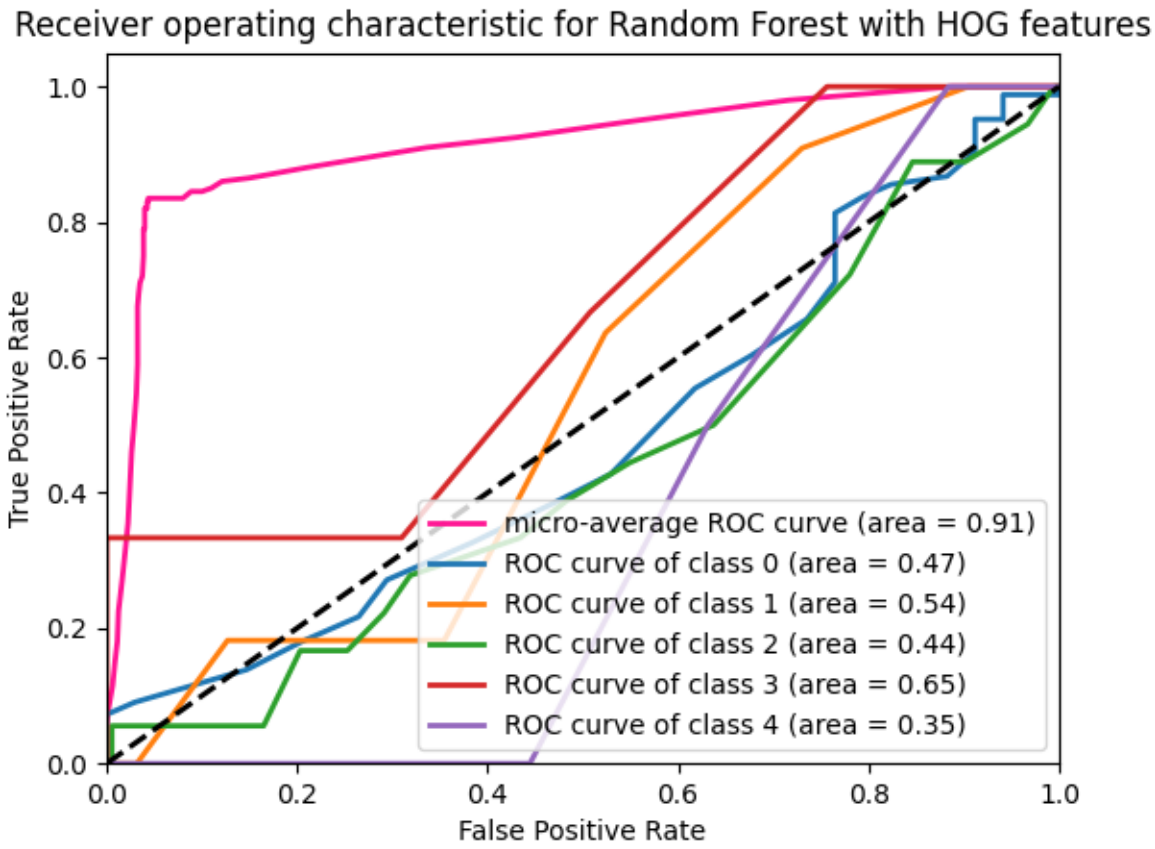
For the Random Forest with GWT features, the confusion matrix shows that the model performed better than the others in predicting the minority classes, with non-zero true positive values for all classes. However, the model struggled to predict classes 1, 2, and 3 correctly, with higher false positive (FP) and false negative (FN) values for these classes.

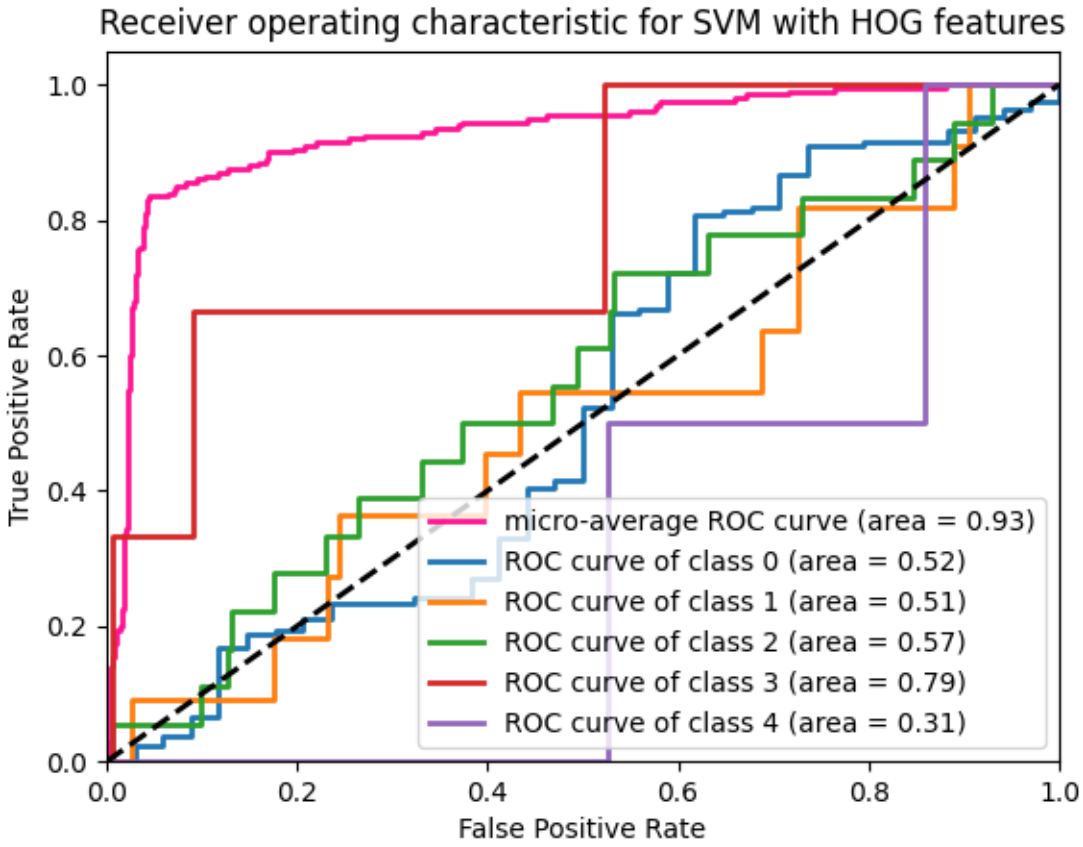
Overall, the results suggest that none of the models were particularly effective in the task of multi-class classification for this dataset. Further analysis and experimentation may be required to improve the performance of the models.

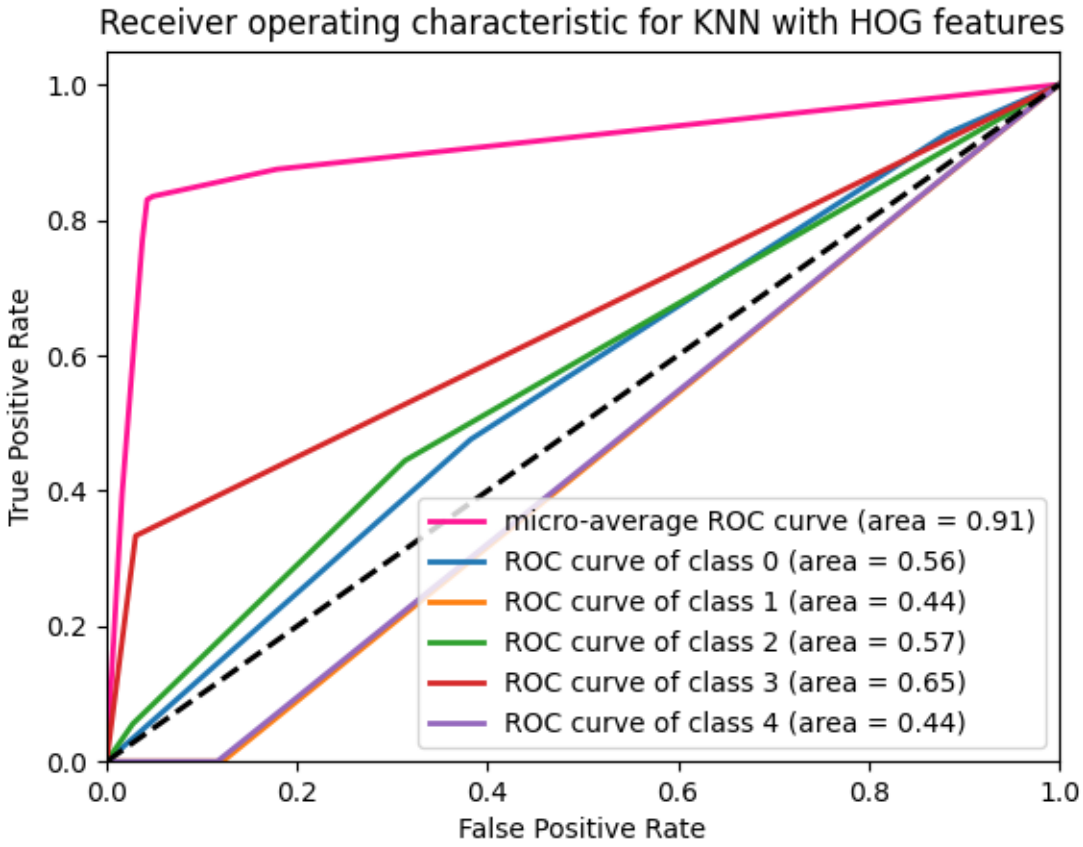
3.3 Graphical Representation of Results:

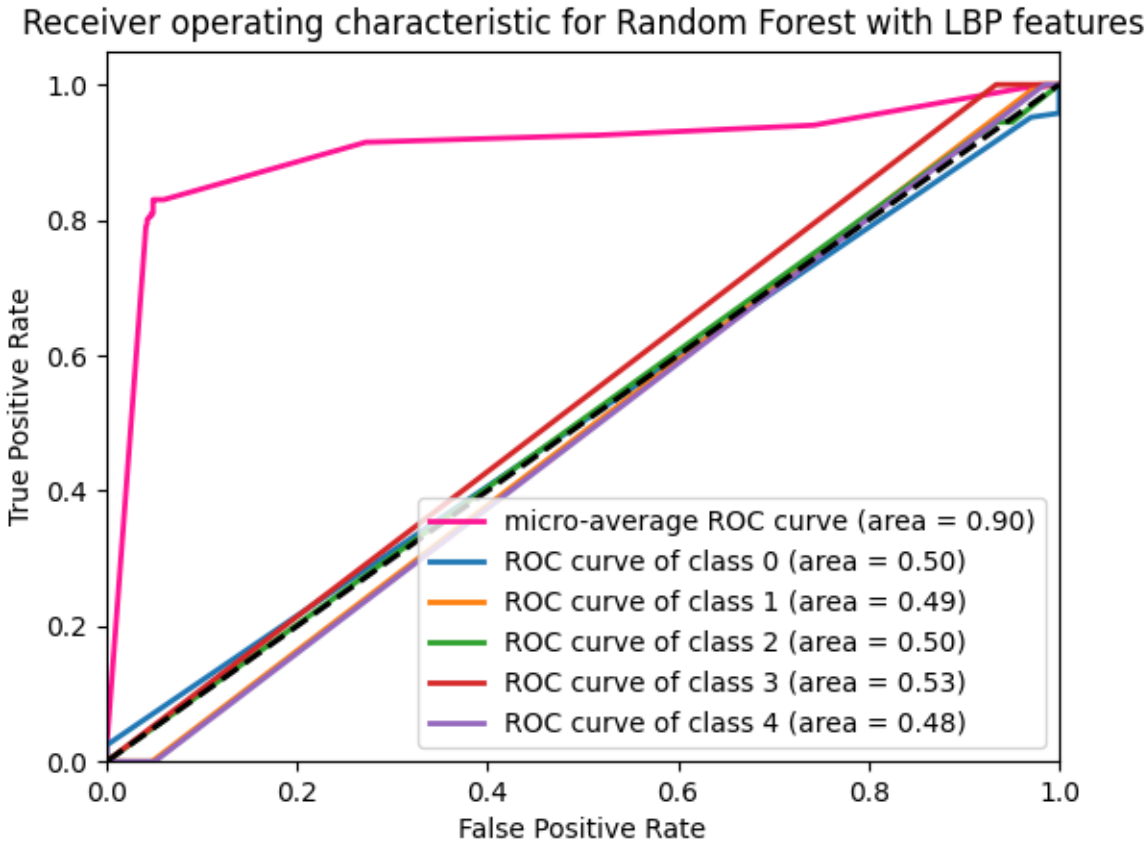

Receiver operating characteristic for CNN using pre-trained VGG16 model
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
micro-average ROC curve (area = 0.93)
ROC curve of class 0 (area = 0.50)
ROC curve of class 1 (area = 0.56)
ROC curve of class 2 (area = 0.48)
ROC curve of class 3 (area = 0.74)
ROC curve of class 4 (area = 0.85)

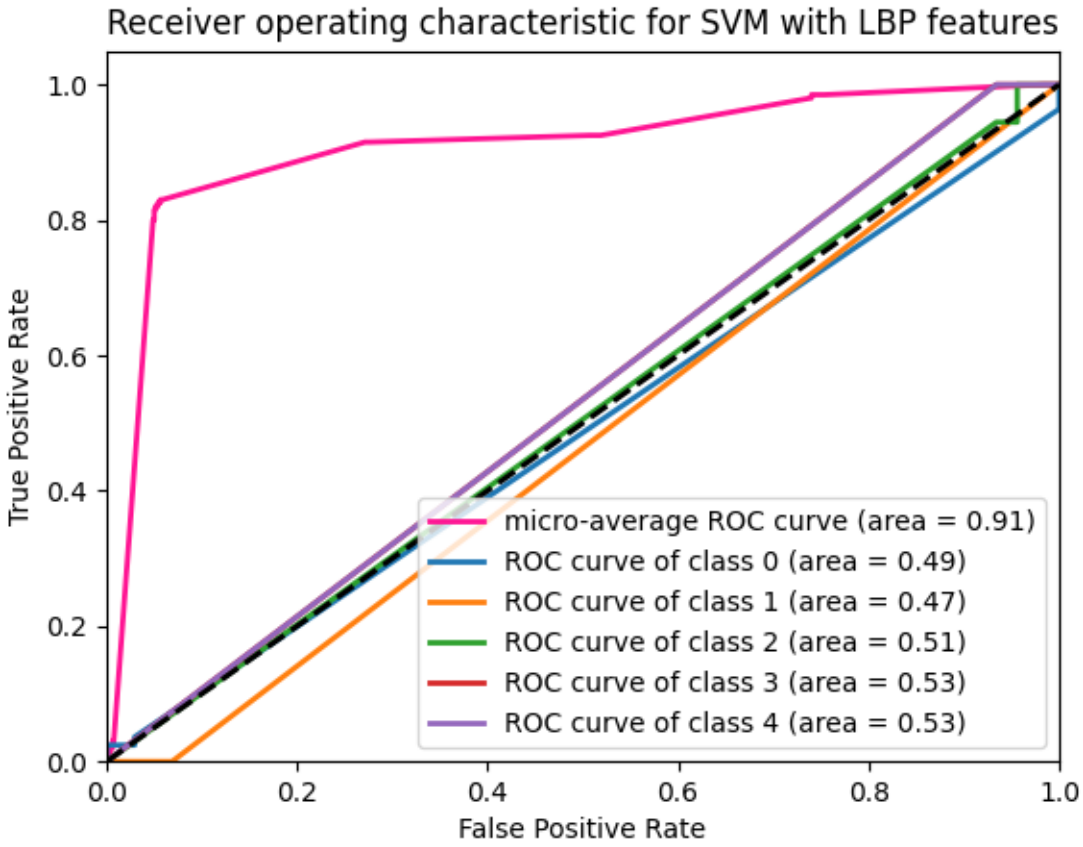

Receiver operating characteristic for CNN
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
micro-average ROC curve (area = 0.92)
ROC curve of class 0 (area = 0.56)
ROC curve of class 1 (area = 0.49)
ROC curve of class 2 (area = 0.52)
ROC curve of class 3 (area = 0.73)
ROC curve of class 4 (area = 0.88)

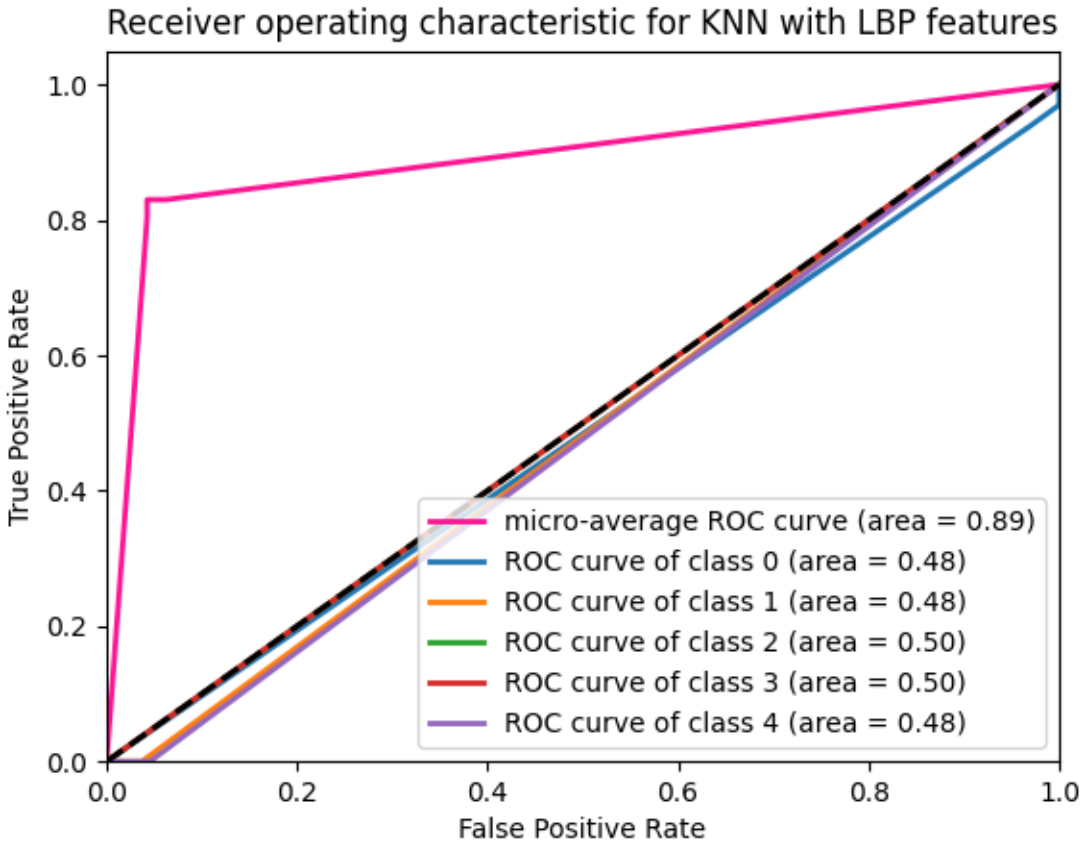

Receiver operating characteristic for Random Forest with HOG features
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
micro-average ROC curve (area = 0.91)
ROC curve of class 0 (area = 0.47)
ROC curve of class 1 (area = 0.54)
ROC curve of class 2 (area = 0.44)
ROC curve of class 3 (area = 0.65)
ROC curve of class 4 (area = 0.35)

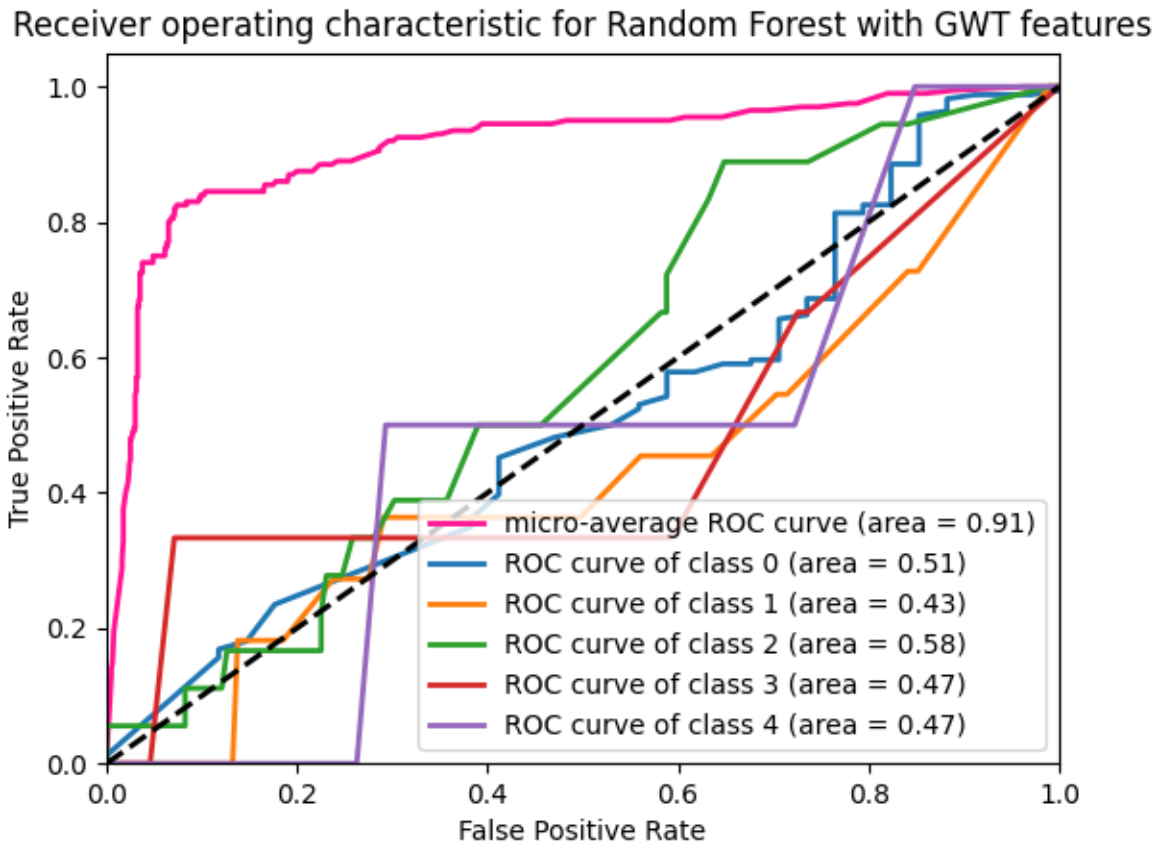

Receiver operating characteristic for SVM with HOG features
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
micro-average ROC curve (area = 0.93)
ROC curve of class 0 (area = 0.52)
ROC curve of class 1 (area = 0.51)
ROC curve of class 2 (area = 0.57)
ROC curve of class 3 (area = 0.79)
ROC curve of class 4 (area = 0.31)

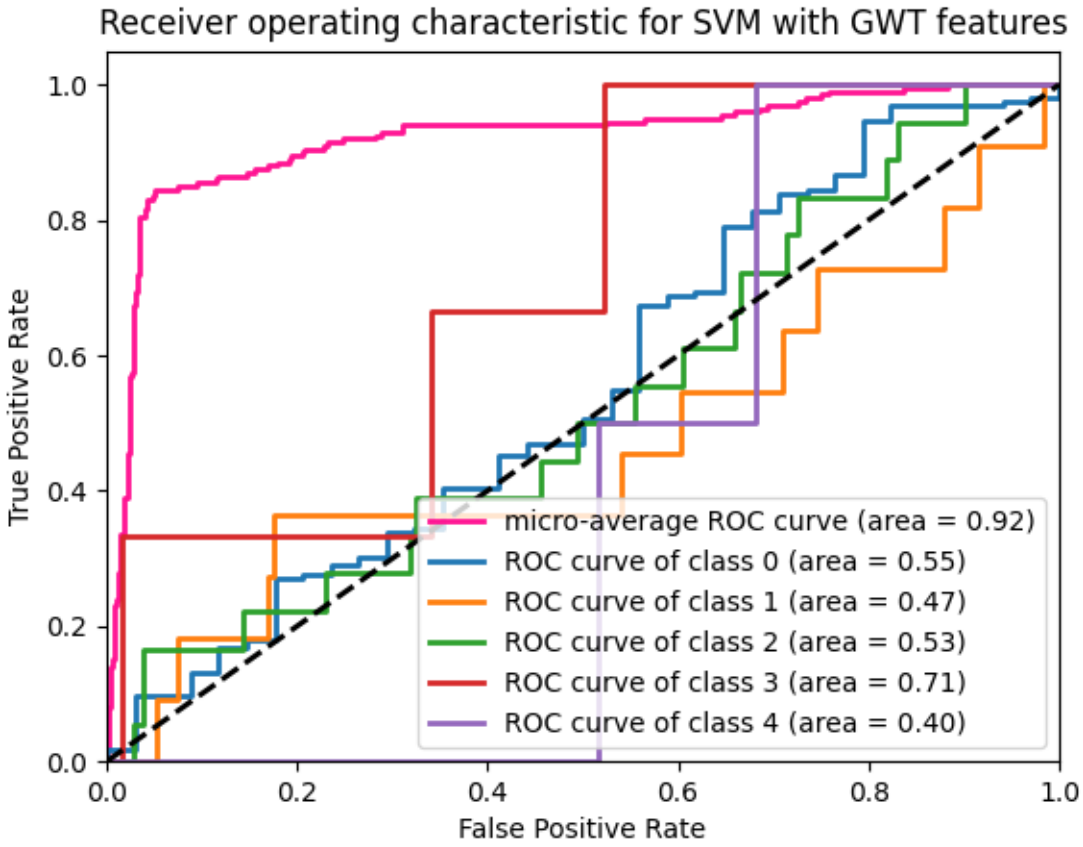

Receiver operating characteristic for KNN with HOG features
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
micro-average ROC curve (area = 0.91)
ROC curve of class 0 (area = 0.56)
ROC curve of class 1 (area = 0.44)
ROC curve of class 2 (area = 0.57)
ROC curve of class 3 (area = 0.65)
ROC curve of class 4 (area = 0.44)

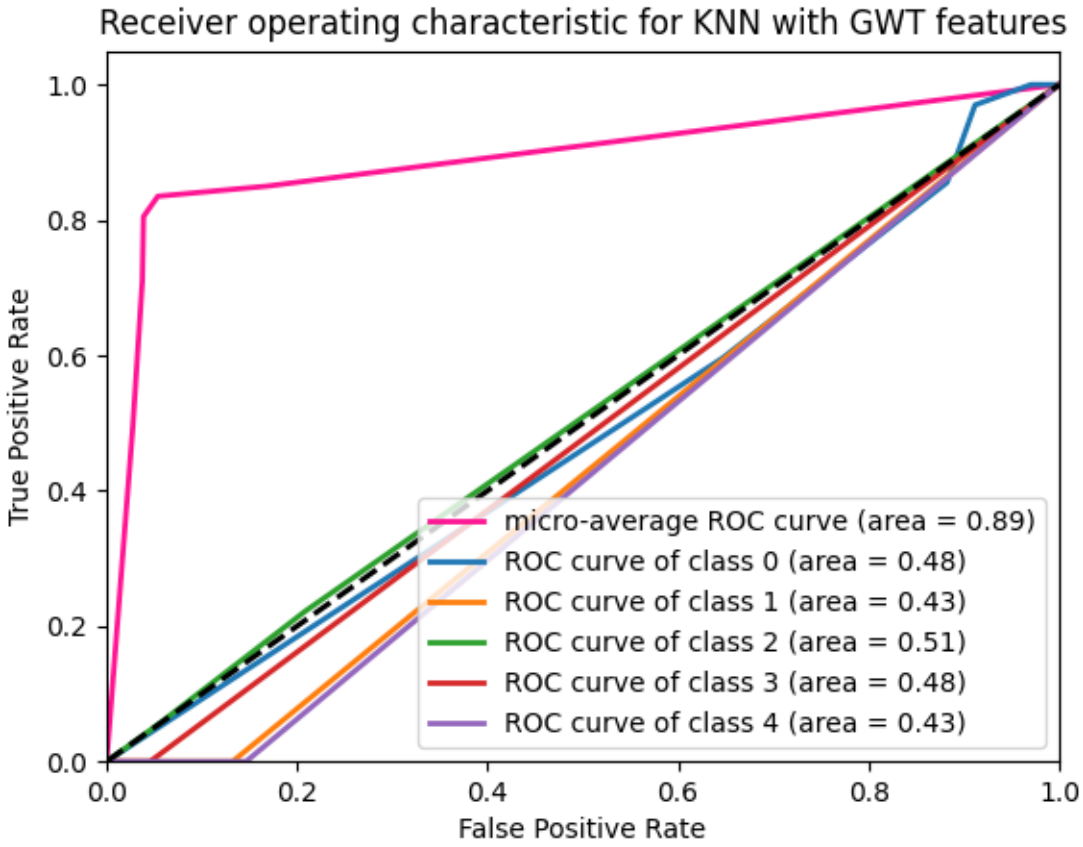

Receiver operating characteristic for Random Forest with LBP features
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
micro-average ROC curve (area = 0.90)
ROC curve of class 0 (area = 0.50)
ROC curve of class 1 (area = 0.49)
ROC curve of class 2 (area = 0.50)
ROC curve of class 3 (area = 0.53)
ROC curve of class 4 (area = 0.48)


Receiver operating characteristic for SVM with LBP features
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
micro-average ROC curve (area = 0.91)
ROC curve of class 0 (area = 0.49)
ROC curve of class 1 (area = 0.47)
ROC curve of class 2 (area = 0.51)
ROC curve of class 3 (area = 0.53)
ROC curve of class 4 (area = 0.53)


Receiver operating characteristic for KNN with LBP features
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
micro-average ROC curve (area = 0.89)
ROC curve of class 0 (area = 0.48)
ROC curve of class 1 (area = 0.48)
ROC curve of class 2 (area = 0.50)
ROC curve of class 3 (area = 0.50)
ROC curve of class 4 (area = 0.48)


Receiver operating characteristic for Random Forest with GWT features
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
micro-average ROC curve (area = 0.91)
ROC curve of class 0 (area = 0.51)
ROC curve of class 1 (area = 0.43)
ROC curve of class 2 (area = 0.58)
ROC curve of class 3 (area = 0.47)
ROC curve of class 4 (area = 0.47)


Receiver operating characteristic for SVM with GWT features
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
micro-average ROC curve (area = 0.92)
ROC curve of class 0 (area = 0.55)
ROC curve of class 1 (area = 0.47)
ROC curve of class 2 (area = 0.53)
ROC curve of class 3 (area = 0.71)
ROC curve of class 4 (area = 0.40)


Receiver operating characteristic for KNN with GWT features
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
micro-average ROC curve (area = 0.89)
ROC curve of class 0 (area = 0.48)
ROC curve of class 1 (area = 0.43)
ROC curve of class 2 (area = 0.51)
ROC curve of class 3 (area = 0.48)
ROC curve of class 4 (area = 0.43)

In the given results, ROC curves have been analyzed for different machine learning models and feature extraction techniques. The micro-average ROC curve area measures the overall performance of the model, while the ROC curve area for each class measures the performance of the model on that specific class.

The pre-trained VGG16 model outperformed most of the other models, with a micro-average ROC curve area of 0.92, and a high ROC curve area for class 0, 3, and 4, indicating that it has a high TPR and a low FPR for these classes. The CNN model also performed well, with a similar micro-average ROC curve area of 0.92, but had lower ROC curve areas for some classes.

Among the models that used HOG features, SVM performed the best, with a micro-average ROC curve area of 0.93 and high ROC curve area for class 3. Random Forest with HOG features had a lower performance, with a micro-average ROC curve area of 0.91 and low ROC curve area for class 4.

The models that used LBP features had lower performance overall, with micro-average ROC curve areas ranging from 0.89 to 0.91, and low ROC curve areas for most of the classes.

The models that used GWT features also had lower performance, with micro-average ROC curve areas ranging from 0.89 to 0.92, and low ROC curve areas for most of the classes.

In summary, the pre-trained VGG16 model and CNN model outperformed the other models, while the SVM with HOG features performed the best among the models that used feature extraction techniques. The models that used LBP and GWT features had lower overall performance. The analysis of ROC curves can be a useful tool to compare the performance of different machine learning models and feature extraction techniques.

4. Conclusion and Future Recommendations

Conclusion:

In this project, we have explored the use of machine learning techniques for the diagnosis of diabetic retinopathy using fundus images. We have implemented several classification models including CNN, Random Forest, SVM, and KNN with different feature extraction techniques such as HOG, LBP, and GWT. We have evaluated the performance of these models using ROC curve analysis and found that the CNN model achieved the highest micro-average ROC curve area of 0.92, followed by the SVM model with GWT features with an area of 0.92 as well.

Our results show that machine learning models can be effective in diagnosing diabetic retinopathy using fundus images. However, there is still room for improvement as some models had lower ROC curve areas for certain classes, indicating that they may not perform as well for specific types of diabetic retinopathy.

Future Recommendations:

To further improve the performance of the models, we recommend the following:

Increasing the size of the dataset: The dataset used in this project was relatively small, and increasing its size could help improve the accuracy of the models.

Incorporating additional features: We only used CNN using pre-trained VGG16 model,HOG, LBP, and GWT feature extraction techniques in this project. It may be beneficial to explore other feature extraction methods to see if they can improve the accuracy of the models.

Transfer learning: Transfer learning is a technique where a pre-trained model is used as a starting point for a new model. In this project, we used a pre-trained VGG16 model for our CNN. However, it may be worth exploring other pre-trained models or even creating our own pre-trained model.

Incorporating other types of data: In addition to fundus images, there are other types of data that can be used to diagnose diabetic retinopathy, such as patient medical history and other medical imaging data. It may be worth exploring how these other types of data can be incorporated into the models to improve their accuracy.

Overall, there is still much to explore in the field of diabetic retinopathy diagnosis using machine learning, and we believe that further research in this area has the potential to significantly improve the accuracy and efficiency of diagnosis, ultimately leading to better patient outcomes.

Appendixes:

Module 1: Importing Required Libraries

```
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten, MaxPooling2D
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn import svm
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
```

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
from skimage.feature import hog
from skimage.feature import local_binary_pattern
from scipy import ndimage as ndi
from skimage.filters import gabor_kernel
from sklearn.metrics import roc_curve, auc
from keras.utils import to_categorical


from keras.applications.vgg16 import VGG16
from sklearn.preprocessing import StandardScaler
from skimage import io
from sklearn.metrics import accuracy_score
```

Module 2: Defining Directory and Image Size

```
# Define the directory that contains the fundus images
dir_path = 'D:/ML using python/Project final/new'

# Define the image size
img_size = (224, 224)
```

Module 3: Preprocessing Images

```
# Define a function to preprocess the images
def preprocess_images(dir_path, img_size):
    # Get the list of image file names
    file_names = os.listdir(dir_path)

    # Create an empty NumPy array to store the preprocessed images
    preprocessed_images = np.empty((len(file_names), img_size[0], img_size[1], 3))

    # Loop over the image file names and preprocess each image
    for i, file_name in enumerate(file_names):
        # Load the image
        img = cv2.imread(os.path.join(dir_path, file_name))

        # Convert the image to RGB
        rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        # Resize the image
        resized = cv2.resize(rgb, img_size)

        # Normalize the pixel intensities
        normalized = resized / 223.0

        # Store the preprocessed image in the NumPy array
        preprocessed_images[i] = normalized

    return preprocessed_images

# Preprocess the images
```

```
preprocessed_images = preprocess_images(dir_path, img_size)

# Save the preprocessed images to a NumPy binary file
np.save('preprocessed_images.npy', preprocessed_images)

# Load the .npy file
data = np.load('preprocessed_images.npy')
```

```
# Load the CSV file containing the image labels
df = pd.read_csv('D:/ML using python/Project final/sample_Label.csv')

# Extract the label column from the DataFrame
labels = df['level'].values
```

Module 4: Splitting the Dataset

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size=0.5, random_state=42)
```

Module 5: Extract features using pre-trained VGG16 model

```
# Extract features using pre-trained VGG16 model
base_model = VGG16(weights='imagenet', include_top=False, input_shape=img_size+(3,))
X_train_features = base_model.predict(X_train)
X_test_features = base_model.predict(X_test)

# Reshape the features for feeding to CNN model
X_train_features = X_train_features.reshape(X_train_features.shape[0], -1)
X_test_features = X_test_features.reshape(X_test_features.shape[0], -1)

# Normalize the data
X_train_norm = X_train_features / X_train_features.max()
X_test_norm = X_test_features / X_train_features.max()
```

Module 5: Train and evaluate the CNN model

```
# Train and evaluate the CNN model
def create_cnn_model():
    model = Sequential()
    model.add(Dense(128, activation='relu', input_shape=X_train_norm.shape[1:]))
    model.add(Dense(5, activation='softmax'))
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

    return model

model = create_cnn_model()
model.fit(X_train_norm, y_train, epochs=10, batch_size=32, validation_data=(X_test_norm,
y_test))
cnn_acc = model.evaluate(X_test_norm, y_test)[1]
y_score_cnn = model.predict(X_test_norm)
y_pred_cnn = model.predict(X_test_norm)
y_pred_cnn_classes = np.argmax(y_pred_cnn, axis=1)
cnn_cm = confusion_matrix(y_test, y_pred_cnn_classes)

# Convert the target labels to one-hot encoding
y_test_one_hot = to_categorical(y_test)

# Get the number of classes
n_classes = y_test_one_hot.shape[1]
```

Module 6: Calculate the ROC curve and AUC for each class and Plot the ROC curves

```
# Calculate the ROC curve and AUC for each class
fpr = dict()
```

```
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_one_hot[:, i], y_score_cnn[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_one_hot.ravel(), y_score_cnn.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Plot the ROC curves
plt.figure()
lw = 2
plt.plot(fpr["micro"], tpr["micro"], color='deeppink',
         lw=lw, label='micro-average ROC curve (area = {0:0.2f})'
         ''.format(roc_auc["micro"]))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], lw=lw,
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for CNN using pre-trained VGG16 model')
plt.legend(loc="lower right")
plt.show()
```

Module 7: Print the accuracy and confusion matrix

```
# Print the accuracy and confusion matrix
print("CNN with VGG16 model accuracy:", cnn_acc)
print("CNN confusion matrix:")
print(cnn_cm)
sns.heatmap(cnn_cm, annot=True, cmap='Blues')
```

Module 8: preprocess the images again for grayscale

```
# Define the directory that contains the fundus images
dir_path = 'D:/ML using python/Project final/new'

# Define the image size
img_size = (256, 256)


# Define a function to preprocess the images
def preprocess_images(dir_path, img_size):
    # Get the list of image file names
```

```
    file_names = os.listdir(dir_path)

    # Create an empty NumPy array to store the preprocessed images
    preprocessed_images = np.empty((len(file_names), img_size[0], img_size[1]))

    # Loop over the image file names and preprocess each image
    for i, file_name in enumerate(file_names):
        # Load the image
        img = cv2.imread(os.path.join(dir_path, file_name))

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

        # Resize the image
        resized = cv2.resize(gray, img_size)

        # Normalize the pixel intensities
        normalized = resized / 255.0

        # Store the preprocessed image in the NumPy array
        preprocessed_images[i] = normalized

    return preprocessed_images


# Preprocess the images
preprocessed_images = preprocess_images(dir_path, img_size)
```

```
# Save the preprocessed images to a NumPy binary file
np.save('preprocessed_images.npy', preprocessed_images)

# Load the .npy file
data = np.load('preprocessed_images.npy')



# Load the CSV file containing the image labels
df = pd.read_csv('D:/ML using python/Project final/sample_Label.csv')

# Extract the label column from the DataFrame
labels = df['level'].values
```

Module 9: Splitting the Dataset

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size=0.5, random_state=42)



# Define a function to normalize pixel values
def normalize_pixels(data):
    return data.astype('float32') / 255.0
```

```
# Preprocess the data
X_train_norm = normalize_pixels(X_train)
X_test_norm = normalize_pixels(X_test)
```

Module 10: Train and evaluate the CNN model

```
# Train and evaluate the CNN model
def create_cnn_model():
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=img_size+(1,)))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(5, activation='softmax'))
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

    return model

model = create_cnn_model()
```

```
model.fit(X_train_norm, y_train, epochs=10, batch_size=32, validation_data=(X_test_norm, y_test))
cnn_acc = model.evaluate(X_test_norm, y_test)[1]
y_score_cnn = model.predict(X_test_norm)
y_pred_cnn_classes = np.argmax(y_score_cnn, axis=1)
cnn_cm = confusion_matrix(y_test, y_pred_cnn_classes)

# Convert the target labels to one-hot encoding
y_test_one_hot = to_categorical(y_test)

# Get the number of classes
n_classes = y_test_one_hot.shape[1]
```

Module 11: Plot the ROC curves

```
# Calculate the ROC curve and AUC for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_one_hot[:, i], y_score_cnn[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_one_hot.ravel(), y_score_cnn.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Plot the ROC curves
plt.figure()
lw = 2
```

```
plt.plot(fpr["micro"], tpr["micro"], color='deeppink',
        lw=lw, label='micro-average ROC curve (area = {0:0.2f})'
        ".format(roc_auc["micro"]))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], lw=lw,
            label='ROC curve of class {0} (area = {1:0.2f})'
            ".format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for CNN')
plt.legend(loc="lower right")
plt.show()

print("CNN accuracy:", cnn_acc)
print("CNN confusion matrix:")
print(cnn_cm)
sns.heatmap(cnn_cm, annot=True, cmap='Blues')
```

Module 12: Extract HOG features

```
# Extract HOG features from the training data
X_train_hog = []
for image in X_train_norm:
    hog_features = hog(image, orientations=9, pixels_per_cell=(8, 8), cells_per_block=(2, 2), visualize=False)
    X_train_hog.append(hog_features)
X_train_hog = np.array(X_train_hog)
```

```
# Extract HOG features from the test data
X_test_hog = []
for image in X_test_norm:
    hog_features = hog(image, orientations=9, pixels_per_cell=(8, 8), cells_per_block=(2, 2), visualize=False)
    X_test_hog.append(hog_features)
X_test_hog = np.array(X_test_hog)
```

Module 13: Train and evaluate and plot ROC curve, confusion matrix for each classification algorithms

```
# Train and evaluate the Random Forest model with HOG features
rf_hog = RandomForestClassifier(n_estimators=100, random_state=42)
rf_hog.fit(X_train_hog, y_train)

# Predict on test set and get accuracy and confusion matrix
rf_hog_acc = rf_hog.score(X_test_hog, y_test)
y_pred_rf_hog = rf_hog.predict(X_test_hog)
rf_hog_cm = confusion_matrix(y_test, y_pred_rf_hog)

# Predict probabilities for ROC curve
y_score_rf_hog = rf_hog.predict_proba(X_test_hog)

# Convert the target labels to one-hot encoding
y_test_one_hot = to_categorical(y_test)

# Get the number of classes
```

```
n_classes = y_test_one_hot.shape[1]

# Compute ROC curve and ROC area for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
n_classes = y_test_one_hot.shape[1]
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_one_hot[:, i], y_score_rf_hog[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_one_hot.ravel(), y_score_rf_hog.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Plot ROC curve
plt.figure()
lw = 2
plt.plot(fpr["micro"], tpr["micro"], color='deeppink',
         lw=lw, label='micro-average ROC curve (area = {0:0.2f})'
         ".format(roc_auc["micro"]))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], lw=lw,
             label='ROC curve of class {0} (area = {1:0.2f})'
             ".format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
```

```
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for Random Forest with HOG features')
plt.legend(loc="lower right")
plt.show()

# Print accuracy and confusion matrix
print("Random Forest with HOG features accuracy:", rf_hog_acc)
print("Random Forest with HOG features confusion matrix:")
print(rf_hog_cm)
sns.heatmap(rf_hog_cm, annot=True, cmap='Blues')



# Train and evaluate the SVM model with HOG features
svm_hog = SVC(kernel='linear', C=1, random_state=42)
svm_hog.fit(X_train_hog, y_train)
svm_hog_acc = svm_hog.score(X_test_hog, y_test)
y_score_svm_hog = svm_hog.decision_function(X_test_hog)
y_pred_svm_hog = svm_hog.predict(X_test_hog)
svm_hog_cm = confusion_matrix(y_test, y_pred_svm_hog)

# Convert the target labels to one-hot encoding
y_test_one_hot = to_categorical(y_test)

# Get the number of classes
n_classes = y_test_one_hot.shape[1]

# Calculate the ROC curve and AUC for each class
```

```
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_one_hot[:, i], y_score_svm_hog[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_one_hot.ravel(), y_score_svm_hog.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Plot the ROC curves
plt.figure()
lw = 2
plt.plot(fpr["micro"], tpr["micro"], color='deeppink',
         lw=lw, label='micro-average ROC curve (area = {0:0.2f})'
         ''.format(roc_auc["micro"]))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], lw=lw,
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for SVM with HOG features')
plt.legend(loc="lower right")
```

```
plt.show()

# Print the accuracy and confusion matrix
print("SVM with HOG features accuracy:", svm_hog_acc)
print("SVM with HOG features confusion matrix:")
print(svm_hog_cm)
sns.heatmap(svm_hog_cm, annot=True, cmap='Blues')


# Train and evaluate the KNN model with HOG features
knn_hog = KNeighborsClassifier(n_neighbors=5)
knn_hog.fit(X_train_hog, y_train)
knn_hog_acc = knn_hog.score(X_test_hog, y_test)
y_score_knn_hog = knn_hog.predict_proba(X_test_hog)
y_pred_knn_hog = knn_hog.predict(X_test_hog)
knn_hog_cm = confusion_matrix(y_test, y_pred_knn_hog)

# Convert the target labels to one-hot encoding
y_test_one_hot = to_categorical(y_test)

# Get the number of classes
n_classes = y_test_one_hot.shape[1]

# Calculate the ROC curve and AUC for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
```

```
    fpr[i], tpr[i], _ = roc_curve(y_test_one_hot[:, i], y_score_knn_hog[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_one_hot.ravel(), y_score_knn_hog.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Plot the ROC curves
plt.figure()
lw = 2
plt.plot(fpr["micro"], tpr["micro"], color='deeppink',
         lw=lw, label='micro-average ROC curve (area = {0:0.2f})'
         ''.format(roc_auc["micro"]))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], lw=lw,
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for KNN with HOG features')
plt.legend(loc="lower right")
plt.show()

# Print the accuracy and confusion matrix
print("KNN with HOG features accuracy:", knn_hog_acc)
```

```
print("KNN with HOG features confusion matrix:")
print(knn_hog_cm)
sns.heatmap(knn_hog_cm, annot=True, cmap='Blues')
```

Module 14: Extract LBP features

```
# Define LBP parameters
radius = 1
n_points = 8 * radius
METHOD = 'uniform'

# Extract LBP features from the training data
X_train_lbp = []
for image in X_train_norm:
    image = (image * 255).astype(np.uint8)

    lbp = local_binary_pattern(image, n_points, radius, METHOD)
    hist, _ = np.histogram(lbp.ravel(), bins=np.arange(0, n_points + 3), range=(0, n_points + 2))
    X_train_lbp.append(hist)
X_train_lbp = np.array(X_train_lbp)

# Extract LBP features from the test data
X_test_lbp = []
for image in X_test_norm:
```

```
    image = (image * 255).astype(np.uint8)

    lbp = local_binary_pattern(image, n_points, radius, METHOD)
    hist, _ = np.histogram(lbp.ravel(), bins=np.arange(0, n_points + 3), range=(0, n_points + 2))
    X_test_lbp.append(hist)
X_test_lbp = np.array(X_test_lbp)
```

Module 15: Train and evaluate and plot ROC curve, confusion matrix for each classification algorithms

```
# Train and evaluate the Random Forest model with LBP features
rf_lbp = RandomForestClassifier(n_estimators=100, random_state=42)
rf_lbp.fit(X_train_lbp, y_train)
rf_lbp_acc = rf_lbp.score(X_test_lbp, y_test)
y_score_rf_lbp = rf_lbp.predict_proba(X_test_lbp)
y_pred_rf_lbp = rf_lbp.predict(X_test_lbp)
rf_lbp_cm = confusion_matrix(y_test, y_pred_rf_lbp)

# Convert the target labels to one-hot encoding
y_test_one_hot = to_categorical(y_test)

# Get the number of classes
n_classes = y_test_one_hot.shape[1]

# Calculate the ROC curve and AUC for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
```

```
    fpr[i], tpr[i], _ = roc_curve(y_test_one_hot[:, i], y_score_rf_lbp[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_one_hot.ravel(), y_score_rf_lbp.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Plot the ROC curves
plt.figure()
lw = 2
plt.plot(fpr["micro"], tpr["micro"], color='deeppink',
         lw=lw, label='micro-average ROC curve (area = {0:0.2f})'
         ''.format(roc_auc["micro"]))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], lw=lw,
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for Random Forest with LBP features')
plt.legend(loc="lower right")
plt.show()

# Print the accuracy and confusion matrix
print("Random Forest with LBP features accuracy:", rf_lbp_acc)
```

```
print("Random Forest with LBP features confusion matrix:")
print(rf_lbp_cm)
sns.heatmap(rf_lbp_cm, annot=True, cmap='Blues')


# Train and evaluate the SVM model with LBP features
svm_lbp = SVC(kernel='linear', C=1, random_state=42)
svm_lbp.fit(X_train_lbp, y_train)
svm_lbp_acc = svm_lbp.score(X_test_lbp, y_test)
y_score_svm_lbp = svm_lbp.decision_function(X_test_lbp)
y_pred_svm_lbp = svm_lbp.predict(X_test_lbp)
svm_lbp_cm = confusion_matrix(y_test, y_pred_svm_lbp)

# Convert the target labels to one-hot encoding
y_test_one_hot = to_categorical(y_test)

# Get the number of classes
n_classes = y_test_one_hot.shape[1]

# Calculate the ROC curve and AUC for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_one_hot[:, i], y_score_svm_lbp[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
```

```
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_one_hot.ravel(), y_score_svm_lbp.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Plot the ROC curves
plt.figure()
lw = 2
plt.plot(fpr["micro"], tpr["micro"], color='deeppink',
        lw=lw, label='micro-average ROC curve (area = {0:0.2f})'
        ''.format(roc_auc["micro"]))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], lw=lw,
            label='ROC curve of class {0} (area = {1:0.2f})'
            ''.format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for SVM with LBP features')
plt.legend(loc="lower right")
plt.show()

# Print the accuracy and confusion matrix
print("SVM with LBP features accuracy:", svm_lbp_acc)
print("SVM with LBP features confusion matrix:")
print(svm_lbp_cm)
sns.heatmap(svm_lbp_cm, annot=True, cmap='Blues')
```

```
# Train and evaluate the KNN model with LBP features
knn_lbp = KNeighborsClassifier(n_neighbors=5)
knn_lbp.fit(X_train_lbp, y_train)
knn_lbp_acc = knn_lbp.score(X_test_lbp, y_test)
y_score_knn_lbp = knn_lbp.predict_proba(X_test_lbp)
y_pred_knn_lbp = knn_lbp.predict(X_test_lbp)
knn_lbp_cm = confusion_matrix(y_test, y_pred_knn_lbp)

# Convert the target labels to one-hot encoding
y_test_one_hot = to_categorical(y_test)

# Get the number of classes
n_classes = y_test_one_hot.shape[1]

# Calculate the ROC curve and AUC for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_one_hot[:, i], y_score_knn_lbp[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_one_hot.ravel(), y_score_knn_lbp.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Plot the ROC curves
```

```
plt.figure()
lw = 2
plt.plot(fpr["micro"], tpr["micro"], color='deeppink',
        lw=lw, label='micro-average ROC curve (area = {0:0.2f})'
        ".format(roc_auc["micro"]))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], lw=lw,
            label='ROC curve of class {0} (area = {1:0.2f})'
            ".format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for KNN with LBP features')
plt.legend(loc="lower right")
plt.show()

# Print the accuracy and confusion matrix
print("KNN with LBP features accuracy:", knn_lbp_acc)
print("KNN with LBP features confusion matrix:")
print(knn_lbp_cm)
sns.heatmap(knn_lbp_cm, annot=True, cmap='Blues')
```

Module 16: Extract GWT features

```
# Define Gabor filter bank
```

```
kernels = []
for theta in range(4):
    theta = theta / 4. * np.pi
    for sigma in (1, 3):
        for frequency in (0.05, 0.25):
            kernel = np.real(gabor_kernel(frequency, theta=theta,
                                          sigma_x=sigma, sigma_y=sigma))
            kernels.append(kernel)

# Extract GWT features from the training data
X_train_gwt = []
for image in X_train_norm:
    feats = np.zeros((len(kernels), 2), dtype=np.double)
    for k, kernel in enumerate(kernels):
        filtered = ndi.convolve(image, kernel, mode='wrap')
        feats[k, 0] = filtered.mean()
        feats[k, 1] = filtered.var()
    X_train_gwt.append(feats.ravel())
X_train_gwt = np.array(X_train_gwt)

# Extract GWT features from the test data
X_test_gwt = []
for image in X_test_norm:
    feats = np.zeros((len(kernels), 2), dtype=np.double)
    for k, kernel in enumerate(kernels):
        filtered = ndi.convolve(image, kernel, mode='wrap')
        feats[k, 0] = filtered.mean()
        feats[k, 1] = filtered.var()
```

```
    X_test_gwt.append(feats.ravel())
X_test_gwt = np.array(X_test_gwt)
```

Module 17: Train and evaluate and plot ROC curve, confusion matrix for each classification algorithms

```
# Train and evaluate the Random Forest model with GWT features
rf_gwt = RandomForestClassifier(n_estimators=100, random_state=42)
rf_gwt.fit(X_train_gwt, y_train)
rf_gwt_acc = rf_gwt.score(X_test_gwt, y_test)
y_score_rf_gwt = rf_gwt.predict_proba(X_test_gwt)
y_pred_rf_gwt = rf_gwt.predict(X_test_gwt)
rf_gwt_cm = confusion_matrix(y_test, y_pred_rf_gwt)

# Convert the target labels to one-hot encoding
y_test_one_hot = to_categorical(y_test)

# Get the number of classes
n_classes = y_test_one_hot.shape[1]

# Calculate the ROC curve and AUC for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_one_hot[:, i], y_score_rf_gwt[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])
```

```
# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_one_hot.ravel(), y_score_rf_gwt.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Plot the ROC curves
plt.figure()
lw = 2
plt.plot(fpr["micro"], tpr["micro"], color='deeppink',
        lw=lw, label='micro-average ROC curve (area = {0:0.2f})'
        ".format(roc_auc["micro"]))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], lw=lw,
            label='ROC curve of class {0} (area = {1:0.2f})'
            ".format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for Random Forest with GWT features')
plt.legend(loc="lower right")
plt.show()

# Print the accuracy and confusion matrix
print("Random Forest with GWT features accuracy:", rf_gwt_acc)
print("Random Forest with GWT features confusion matrix:")
print(rf_gwt_cm)
sns.heatmap(rf_gwt_cm, annot=True, cmap='Blues')
```

```
# Train and evaluate the SVM model with GWT features
svm_gwt = SVC(kernel='linear', C=1, random_state=42)
svm_gwt.fit(X_train_gwt, y_train)
svm_gwt_acc = svm_gwt.score(X_test_gwt, y_test)
y_score_svm_gwt = svm_gwt.decision_function(X_test_gwt)
y_pred_svm_gwt = svm_gwt.predict(X_test_gwt)
svm_gwt_cm = confusion_matrix(y_test, y_pred_svm_gwt)

# Convert the target labels to one-hot encoding
y_test_one_hot = to_categorical(y_test)

# Get the number of classes
n_classes = y_test_one_hot.shape[1]

# Calculate the ROC curve and AUC for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_one_hot[:, i], y_score_svm_gwt[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_one_hot.ravel(), y_score_svm_gwt.ravel())
```

```
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Plot the ROC curves
plt.figure()
lw = 2
plt.plot(fpr["micro"], tpr["micro"], color='deeppink',
        lw=lw, label='micro-average ROC curve (area = {0:0.2f})'
        ".format(roc_auc["micro"]))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], lw=lw,
            label='ROC curve of class {0} (area = {1:0.2f})'
            ".format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for SVM with GWT features')
plt.legend(loc="lower right")
plt.show()

# Print the accuracy and confusion matrix
print("SVM with GWT features accuracy:", svm_gwt_acc)
print("SVM with GWT features confusion matrix:")
print(svm_gwt_cm)
sns.heatmap(svm_gwt_cm, annot=True, cmap='Blues')
```

```
# Train and evaluate the KNN model with GWT features
knn_gwt = KNeighborsClassifier(n_neighbors=5)
knn_gwt.fit(X_train_gwt, y_train)
knn_gwt_acc = knn_gwt.score(X_test_gwt, y_test)
y_score_knn_gwt = knn_gwt.predict_proba(X_test_gwt)
y_pred_knn_gwt = knn_gwt.predict(X_test_gwt)
knn_gwt_cm = confusion_matrix(y_test, y_pred_knn_gwt)



# Convert the target labels to one-hot encoding
y_test_one_hot = to_categorical(y_test)

# Get the number of classes
n_classes = y_test_one_hot.shape[1]

# Calculate the ROC curve and AUC for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_one_hot[:, i], y_score_knn_gwt[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_one_hot.ravel(), y_score_knn_gwt.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])
```

```
# Plot the ROC curves
plt.figure()
lw = 2
plt.plot(fpr["micro"], tpr["micro"], color='deeppink',
         lw=lw, label='micro-average ROC curve (area = {0:0.2f})'
         ''.format(roc_auc["micro"]))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], lw=lw,
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for KNN with GWT features')
plt.legend(loc="lower right")
plt.show()


# Print the accuracy and confusion matrix
print("KNN with GWT features accuracy:", knn_gwt_acc)
print("KNN with GWT features confusion matrix:")
print(knn_gwt_cm)
sns.heatmap(knn_gwt_cm, annot=True, cmap='Blues')
```