SIMIYU PATIENCE SIMULI
ENE221-0110/2018

ETI 2507: DIGITAL IMAGE PROCESSING
LAB 1 & LAB 2

Abstract
Defect detection is a pivotal aspect of quality control in manufacturing. Extensive research has been dedicated to implementing defect detection systems across various domains, ranging from steel surfaces to fruit grading, with an increasingly prominent application in smart factories. The textile manufacturing process, in particular, has embraced automated fabric defect detection systems due to the inherent challenge of discerning minuscule imperfections within intricate fabric textures. Traditional manual inspection methods suffer from issues of accuracy and escalating labour costs, necessitating the use of automated solutions. In light of these challenges, this project aims to develop a robust system for fabric defect detection utilizing advanced image processing techniques and machine learning models. Through the fusion of these innovative methodologies, we seek to address these challenges comprehensively. The goal is to establish an automated defect detection system capable of enhancing accuracy, reducing labour costs, and contributing to the optimization of quality control processes in the textile manufacturing sector.

Introduction

Manufacturing has undergone numerous transformative shifts propelled by technological advancements, with an intensified focus on precision and quality control. Within this context, defect detection emerges as a critical aspect that helps safeguard the integrity of diverse products across industries. As manufacturing processes evolve towards automation and smart systems, the textile industry grapples with a unique challenge - the intricate nature of fabric textures demands nuanced defect detection methodologies.

Historically, manual inspection has been the bedrock of quality assurance, but its limitations, including accuracy issues and escalating labour costs, prompt a shift. The need for a more efficient, accurate, and scalable solution has catalyzed the convergence of Image Processing techniques and machine learning models in the pursuit of fabric defect detection. This project delves into this convergence of the two, seeking to develop an automated system that not only navigates the complexities of fabric textures but also augments the efficacy of defect identification. By integrating advanced image processing methodologies and harnessing the learning capabilities of ML models, the project aspires to offer a transformative solution for the textile industry [1].

In the subsequent sections, we navigate the methodologies, techniques, and frameworks that constitute our approach, starting from the Methodologies section, then to the Discussion section and finally to the Conclusion and References sections.

Objectives

The objectives of this project include:

1. To create an automated system capable of detecting defects in fabric textures.
2. To implement advanced image processing techniques in the spatial and frequency domains to preprocess fabric images.
3. To develop and implement feature extraction methodologies, including Gray-Level Co-occurrence Matrix (GLCM) and Local Binary Pattern (LBP).
4. To integrate machine learning models to harness their power for pattern recognition and classification of fabric defects.
5. To optimize the fabric defect detection system by fine-tuning hyperparameters.

Methodology

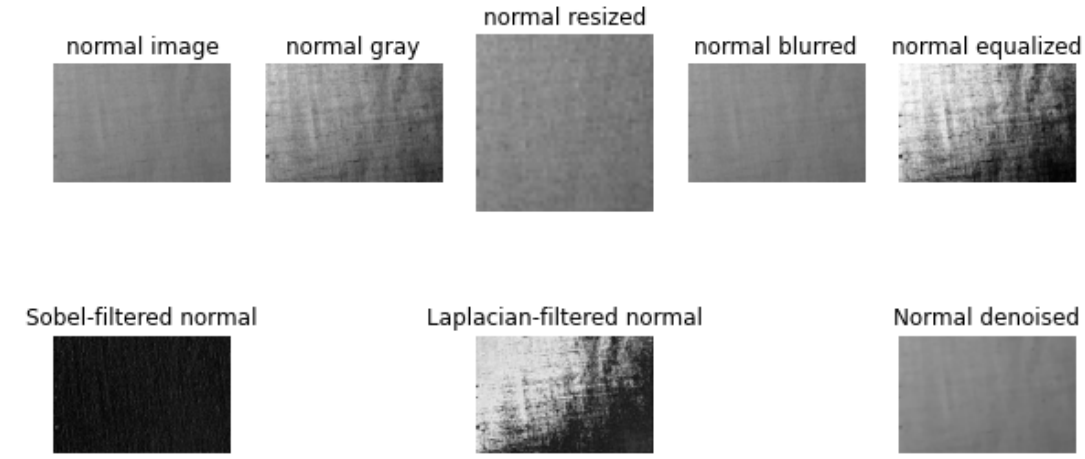
To complete this project, we started with pre-processing in the spatial domain, followed by preprocessing in the frequency domain, then feature extraction, feature labelling, model building, model training, and finally model evaluation and defect detection (prediction). All these processes have been highlighted below.

Pre-processing in the Spatial Domain

```
# NORMAL IMAGE PRE-PROCESSING IN THE SPATIAL DOMAIN

# Convert normal image to grayscale
normal_gray_image = cv2.cvtColor(normal_image, cv2.COLOR_BGR2GRAY)
normal_blurred_image = cv2.GaussianBlur(normal_image, (7, 7), 2)  # Apply Gaussian blur
normal_resized_image = cv2.resize(normal_image, (50, 50))  # Resize the image

#Histogram equalization: Enhance the contrast of the images to improve the visibility of defects
normal_equalized_image = cv2.equalizeHist(normal_gray_image)

#Non-local means denoising
normal_denoised=cv2.fastNlMeansDenoising(normal_image,None,10,7,21)

# Sobel filter kernel
sobel_filter = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1]
], dtype=np.float32)

sobel_image = normal_gray_image

# Apply Sobel filter
sobel_filtered_image = cv2.filter2D(sobel_image, -1, sobel_filter)

filter_kernel = np.array([
    [0, 1, 0],
    [0, -4, 0],
    [0, 1, 0]
], dtype=np.float32)
```

```
filter_kernel = np.array([
    [0, 1, 0],
    [0, -4, 0],
    [0, 1, 0]
], dtype=np.float32)
image = normal_gray_image
filtered_image = np.zeros_like(image, dtype=np.float32)
for i in range(image.shape[0] - 2):
    for j in range(image.shape[1] - 2):
        filtered_image[i, j] = np.sum(image[i:i+3, j:j+3] * filter_kernel)

laplacian_filtered_image = filtered_image.astype(np.uint8)

#Visualization of Normal image vs applied changes
plt.figure(figsize=(10, 5))

plt.subplot(2, 5, 1), plt.imshow(normal_image, cmap='gray'), plt.title('normal image'), plt.axis('off')
plt.subplot(2, 5, 2), plt.imshow(normal_gray_image, cmap='gray'), plt.title('normal gray'), plt.axis('off')
plt.subplot(2, 5, 3), plt.imshow(normal_resized_image, cmap='gray'), plt.title('normal resized'), plt.axis('off')
plt.subplot(2, 5, 4), plt.imshow(normal_blurred_image, cmap='gray'), plt.title('normal blurred'), plt.axis('off')
plt.subplot(2, 5, 5), plt.imshow(normal_equalized_image, cmap='gray'), plt.title('normal equalized'), plt.axis('off')
plt.subplot(2, 5, 6), plt.imshow(sobel_filtered_image, cmap='gray'), plt.title('Sobel-filtered normal'), plt.axis('off')
plt.subplot(2, 5, 8), plt.imshow(laplacian_filtered_image, cmap='gray'), plt.title('Laplacian-filtered normal'), plt.axis('off')
#plt.subplot(2, 6, 10), plt.imshow(normal_equalized_image, cmap='gray'), plt.title('Normal equalized'), plt.axis('off')
plt.subplot(2, 5, 10), plt.imshow(normal_denoised, cmap='gray'), plt.title('Normal denoised'), plt.axis('off')

plt.show()
```

Running the above code resulted in the following:

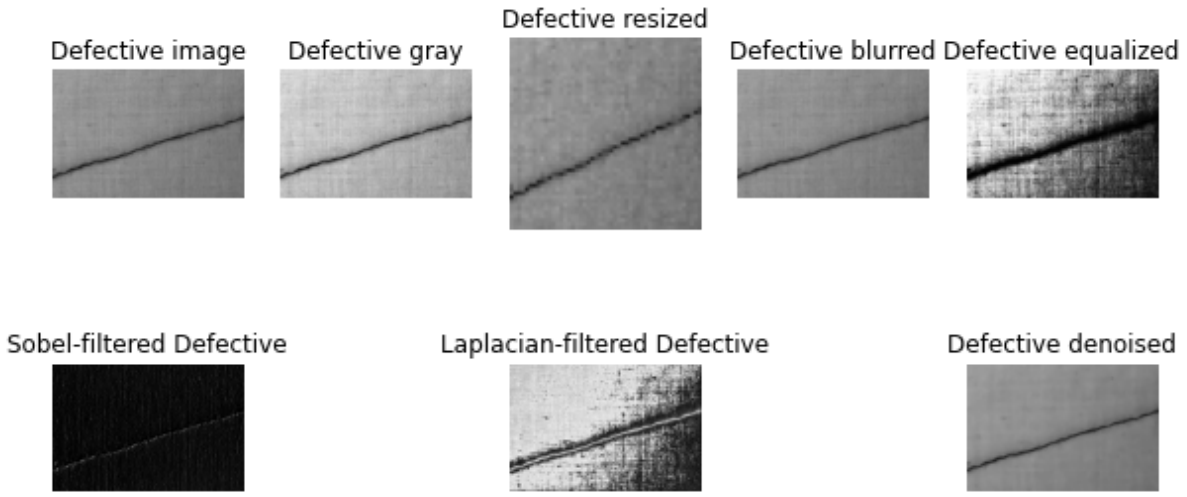


- After that, I did the same for the defective image:

```
# DEFECTIVE IMAGE PRE-PROCESSING IN THE SPATIAL DOMAIN
defective_gray_image = cv2.cvtColor(defective_image, cv2.COLOR_BGR2GRAY)  # Convert to grayscale
defective_resized_image = cv2.resize(defective_image, (50, 50))  # Resize the image
defective_blurred_image = cv2.GaussianBlur(defective_image, (7, 7), 2)  # Apply Gaussian blur

#Histogram equalization: Enhance the contrast of the images to improve the visibility of defects
defective_equalized_image = cv2.equalizeHist(defective_gray_image)

#Non-local means denoising
defective_denoised=cv2.fastNlMeansDenoising(defective_image,None,10,7,21)

# Sobel filter kernel
sobel_filter = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1]
], dtype=np.float32)

defective_sobel_image = defective_gray_image

# Apply Sobel filter
sobel_filtered_defective_image = cv2.filter2D(defective_sobel_image, -1, sobel_filter)

filter_kernel = np.array([
    [0, 1, 0],
    [0, -4, 0],
    [0, 1, 0]
], dtype=np.float32)
image = defective_gray_image
defective_filtered_image = np.zeros_like(image, dtype=np.float32)
```

```
filter_kernel = np.array([
    [0, 1, 0],
    [0, -4, 0],
    [0, 1, 0]
], dtype=np.float32)
image = defective_gray_image
defective_filtered_image = np.zeros_like(image, dtype=np.float32)
for i in range(image.shape[0] - 2):
    for j in range(image.shape[1] - 2):
        filtered_image[i, j] = np.sum(image[i:i+3, j:j+3] * filter_kernel)

laplacian_filtered_defective_image = filtered_image.astype(np.uint8)

#Visualization of Normal image vs applied changes
plt.figure(figsize=(10, 5))

plt.subplot(2, 5, 1), plt.imshow(defective_image, cmap='gray'), plt.title('Defective image'), plt.axis('off')
plt.subplot(2, 5, 2), plt.imshow(defective_gray_image, cmap='gray'), plt.title('Defective gray'), plt.axis('off')
plt.subplot(2, 5, 3), plt.imshow(defective_resized_image, cmap='gray'), plt.title('Defective resized'), plt.axis('off')
plt.subplot(2, 5, 4), plt.imshow(defective_blurred_image, cmap='gray'), plt.title('Defective blurred'), plt.axis('off')
plt.subplot(2, 5, 5), plt.imshow(defective_equalized_image, cmap='gray'), plt.title('Defective equalized'), plt.axis('off')
plt.subplot(2, 5, 6), plt.imshow(sobel_filtered_defective_image, cmap='gray'), plt.title('Sobel-filtered Defective'), plt.axis('
plt.subplot(2, 5, 8), plt.imshow(laplacian_filtered_defective_image, cmap='gray'), plt.title('Laplacian-filtered Defective'), plt
#plt.subplot(2, 5, 8), plt.imshow(defective_equalized_image, cmap='gray'), plt.title('Defective equalized'), plt.axis('off')
plt.subplot(2, 5, 10), plt.imshow(defective_denoised, cmap='gray'), plt.title('Defective denoised'), plt.axis('off')

plt.show()
```

- The results are as follows:



Frequency Domain Pre-processing

- This step started with adjusting the cutoff frequency, D, of the low-pass filter to find the optimum value as indicated by the following code.

```
# Perform 2D Fourier Transform
image = defective_gray_image
f_transform = np.fft.fft2(image)
f_transform_magnitude = np.abs(f_transform)
f_transform_shifted = np.fft.fftshift(f_transform)
f_transform_magnitude_shifted = np.abs(f_transform_shifted)

# Create a low-pass filter
rows, cols = image.shape
crow, ccol = rows // 2, cols // 2
D = 5 # Adjusted to 45, 35, 25, 25, and 5 to find optimum value
H = np.zeros((rows, cols), np.uint8)
H[crow - D:crow + D, ccol - D:ccol + D] = 1
H_HP = 1-H

# Apply the filter to the Fourier Transform
filtered_f_transform_shifted = f_transform_shifted * H_HP
filtered_f_transform_shifted_mag = np.abs(filtered_f_transform_shifted)

# Perform an inverse FFT to return to the spatial domain
filtered_image = np.fft.ifft2(np.fft.ifftshift(filtered_f_transform_shifted))
filtered_image = np.abs(filtered_image).astype(np.uint8)

# Display the original and filtered images
plt.subplot(121), plt.imshow(image, cmap='gray'), plt.title('Original Image')
plt.subplot(122), plt.imshow(filtered_image, cmap='gray'), plt.title('Filtered Image')
plt.show()
```

- Next, I experimented with the high-pass filter using the code displayed below.

```
#Experimenting with the high-pass filter
image_hp = defective_gray_image

rows, cols = image_hp.shape
crow, ccol = rows // 2, cols // 2
D1 = 5

#Create a high-pass filter
F = np.ones((rows, cols), np.uint8)
F[crow - D1:crow + D1, ccol - D1:ccol + D1] = 1
F_HP = 1-H

# Perform 2D Fourier Transform
f_transform_hp = np.fft.fft2(image_hp)
f_transform_shifted_hp = np.fft.fftshift(f_transform_hp)

# Apply the filter to the Fourier Transform
filtered_f_transform_shifted_hp = f_transform_shifted_hp * F_HP
filtered_f_transform_shifted_mag_hp = np.abs(filtered_f_transform_shifted_hp)

#Perform an inverse FFT to return to the spatial domain
filtered_image_hp = np.fft.ifft2(np.fft.ifftshift(filtered_f_transform_shifted_hp))
filtered_image_hp = np.abs(filtered_image_hp).astype(np.uint8)

# Display the original and filtered images
plt.subplot(121), plt.imshow(image_hp, cmap='gray'), plt.title('Original Image')
plt.subplot(122), plt.imshow(filtered_image_hp, cmap='gray'), plt.title('Filtered Image')
plt.show()
```

- The steps above were meant to help with experimenting with different types of image preprocessing and seeing their effects. The next steps involve including these preprocessing steps in the defect detection pipeline.

Loading and Preprocessing steps

- I started by including the preprocessing steps in a function so they can be performed on different images iteratively. The code for this is shown below:

```
# Define a function to load and preprocess images from a folder
def load_and_preprocess(folder_path,label):

# Initialize empty lists to store preprocessed images and labels
    images = []
    labels = []

# Iterate through files in the specified folder
    for filename in os.listdir(folder_path):

# Check if the file has a '.jpg' extension (adjust as needed)
        if filename.endswith('.tif'):  # Adjusted file extension to .tif

            image_path = os.path.join(folder_path, filename) # Construct the full path to the image file
            image = cv2.imread(image_path)  # Read the image using OpenCV

            if image is not None:
                # Extract features (e.g., mean pixel intensity)
                image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
                image_resized = cv2.resize(image_gray, (8, 8))  # Resize the image
                image_blurred = cv2.GaussianBlur(image_resized, (5, 5), 0)  # Apply Gaussian blur
                mean_intensity = np.mean(image_blurred)

                images.append(image_blurred)
                labels.append(label)  # Label for normal images (0) or defective images (1)

    return images, labels
```

- This was followed by designating the folder with normal images and that with defective images. After that, normal images, normal labels, defective images, and defective labels were assigned after calling the load_and_preprocess function on the necessary folders.

Feature Extraction Procedure

- This section started with defining and tuning GLCM properties to ensure they are optimal, followed by normalizing the image values to fit within the specified number of levels, then computing the GLCM properties, and calculating GLCM properties. The code below displays these steps in code.

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, color, feature
from skimage.feature import greycoprops
from skimage.feature import local_binary_pattern
from skimage import data

# Load an example grayscale image
image = normal_gray_image

# Convert the image to grayscale (if it's not already)
if image.ndim > 2:
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)


# Define the GLCM properties
levels = 15 # Increased for more detail in medium-high resolution images
distances = [1] # Including a smaller distance to capture fine defects
angles = [45]  # Multiple angles for potential defect orientations

# Normalize the image values to fit within the specified number of levels
image = (image /255 * (levels - 1)).astype(np.uint8)

# Compute the GLCM
glcm = feature.greycomatrix(image, [1], [0], symmetric=True, normed=True, levels = levels)
print(f"glcm size {glcm.shape}")
```

```
# Calculate GLCM properties
contrast = greycoprops(glcm, 'contrast')
dissimilarity = greycoprops(glcm, 'dissimilarity')
homogeneity = greycoprops(glcm, 'homogeneity')
energy = greycoprops(glcm, 'energy')
correlation = greycoprops(glcm, 'correlation')

# Print the calculated GLCM properties
print(f'Contrast: {contrast}')
print(f'Dissimilarity: {dissimilarity}')
print(f'Homogeneity: {homogeneity}')
print(f'Energy: {energy}')
print(f'Correlation: {correlation}')

# Display the original image and the computed GLCM properties
plt.figure(figsize=(12, 4))

plt.subplot(121)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(122)
plt.imshow(glcm[:, :, 0, 0], cmap='gray', aspect='equal')
plt.title('Gray-Level Co-occurrence Matrix (GLCM)')
plt.axis('off')

plt.show()
```

- Next, I increased the levels to 32 to capture more texture detail in medium-high resolution images, potentially helping to distinguish defects more effectively [2]. After that, I included both distances of 1 and 2 to capture tear characteristics at different scales,

as tears might have varying widths. Following that, I included multiple angles to consider tears in different orientations, as they might not always be perfectly aligned with the image axes [2]. Finally, I leveraged NumPy array operations for efficient GLCM calculation across multiple distances and angles. The code used was as shown below.

- The next feature stage in feature extraction was computing the Local Binary Pattern (LBP) and its histogram for a grayscale image. I increased the radius to 2, intending to capture larger defect patterns in the texture [3]. The number of points (n_points) was then calculated based on the chosen radius. With the LBP image computed, I proceeded to generate a histogram of LBP values. To ensure that the histogram is easily interpretable and comparable across images, I normalized its values [3]. Finally, I printed and visualised the LBP histogram to comprehensively understand the distribution of local binary patterns in the image. The code for this procedure is displayed below.

Feature Labeling

- I defined a Python function named extract_features_from_images to encapsulate the feature extraction logic. This function takes a list of images as input. For each image in the provided list, I iterated through the images, moving to the next step for each image. Then I calculated three different features for each image: Mean Intensity, Variance Intensity, and Standard Deviation Intensity. Additionally, I calculated texture-related features such as entropy, contrast, and dissimilarity using methods like Shannon entropy, GLCM (Grey Level Co-occurrence Matrix), and Local Binary Pattern (LBP). These features provide insights into the statistical and textural characteristics of the images [4]. The combination of these features aims to create a comprehensive representation for subsequent analysis tasks [4]. The code for this is shown below.

```
def extract_features_from_images(images):
    features = []

    for image in images:

        # Calculate statistical features
        mean_intensity = np.mean(image)
        variance_intensity = np.var(image)
        std_intensity = np.std(image)

        # Calculate texture-related features
        entropy_ = shannon_entropy(image)  # Measure of randomness in texture
        contrast = greycoprops(greycomatrix(image, [1], [0], levels=256), 'contrast')[0, 0]  # Local intensity variations
        dissimilarity = greycoprops(greycomatrix(image, [1], [0], levels=256), 'dissimilarity')[0, 0]  # Difference in pixel valu
        lbp_hist, _ = np.histogram(local_binary_pattern(image, 8, 1, method='uniform'), bins=range(0, 257))  # Local texture patt

        # Combine all features
        feature = [mean_intensity, variance_intensity, std_intensity, entropy_, contrast, dissimilarity]
        feature.extend(lbp_hist)  # Append the LBP histogram features
        feature = np.round(feature, 2)
        features.append(feature)

    return features
```

- Next, I began by extracting features from a collection of images using the function named extract_features_from_images. This function calculated various features for each image,

such as mean intensity, variance intensity, standard deviation intensity, entropy, contrast, and dissimilarity. To facilitate further analysis, I organized the extracted features into a NumPy array named samp. This array allowed for convenient manipulation and visualization of the extracted information. For focused analysis, I isolated specific columns from the feature matrix. In this case, I selected the first column (samp_1), representing mean intensity. To prepare the data for visualization, I transposed the array, ensuring compatibility with the plotting process. Finally, I proceeded to visualize the selected feature (mean intensity) across all images using a histogram. The x-axis represents the image index, while the y-axis represents the corresponding feature values.

```
In [88]: all_features = extract_features_from_images(all_images)

In [89]: samp = np.array(all_features)

In [90]: samp_1 = samp[:,0]
         samp_2 = np.transpose(samp_1)

In [93]: samp_1.shape

Out[93]: (100,)

In [96]: y = np.arange(1, 101)

In [97]: # Plot a histogram for the selected feature
         plt.figure(figsize=(8, 6))
         plt.bar(y, samp[:,2], color='b', alpha=0.7)
         plt.xlabel('Image Index')
         plt.ylabel('Feature Value')
         plt.title('Feature Visualization with Image Index')
         plt.grid(axis='y')
         plt.show()
```

Model Building, Training, and Evaluation

- In this section, I built two models, Support Vector Machine (SVM) and K Nearest Neighbor (KNN), which were proven to perform well for defect detection according to the authors of [3] and [5]. After that, I trained both models, evaluated them by testing their accuracy and then adjusted their parameters to view how they changed the accuracy levels. The code snippet below shows these steps.

```
# Import the necessary libraries
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
```

```
# Load labeled features and labels (defect vs. non-defect)
features = all_features
labels = all_labels

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2, random_state=42)
```

```
len(X_train)
```

```
80
```

```
from sklearn.neighbors import KNeighborsClassifier
```

```
# Create a KNN classifier
knn_classifier = KNeighborsClassifier(n_neighbors=3)  # From experiementation, 3 is optimum for accuarcy

# Train the KNN classifier
knn_classifier.fit(X_train, y_train)

# Make predictions on the test set
```

```
# Make predictions on the test set
y_pred = knn_classifier.predict(X_test)

# Calculate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

```
Accuracy: 0.90
```

```
# Create a Support Vector Machine (SVM) classifier
svm_classifier = SVC(C=0.1, kernel='linear', gamma='scale')

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

```
SVC(C=0.1, kernel='linear')
```

```


# Make predictions on the test data
y_pred = svm_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)

# Print the accuracy of the classifier
print(f"Accuracy of the SVM classifier: {accuracy:.2f}")
```

```
Accuracy of the SVM classifier: 0.95
```

```
from sklearn.model_selection import GridSearchCV

# Parameter grid for SVM
param_grid = {'C': [0.1, 1, 10, 100], 'kernel': ['linear', 'rbf', 'poly'], 'gamma': ['scale', 0.001, 0.01, 0.1]}

# Create GridSearchCV object
grid_search = GridSearchCV(svm_classifier, param_grid, cv=5)  # cv=5 for 5-fold cross-validation

# Perform grid search
grid_search.fit(X_train, y_train)

# Get best parameters and model
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_
print(f"Best parameters: {best_params}")
print(f"Best model: {best_model}")

# Print best accuracy
print(f"Best accuracy: {grid_search.best_score_:.2f}")
```

```
Best parameters: {'C': 0.1, 'gamma': 'scale', 'kernel': 'linear'}
Best model: SVC(C=0.1, kernel='linear')
Best accuracy: 0.94
```

Findings and Discussion

The first step for both images was converting them to grayscale. Performing this colour space conversion has a variety of advantages. First, grayscale images have a single intensity channel, as opposed to colour images which typically have three channels (red, green, blue). Converting to grayscale simplifies the representation of the images, making them easier to process [6]. Grayscale images also provide a sufficient representation of texture, and algorithms designed for grayscale images are often simpler and computationally more efficient. Additionally, working with grayscale images reduces the computational complexity of algorithms. Many image processing and computer vision techniques are designed to work efficiently with single-channel images. Also, grayscale images are less sensitive to variations in colour due to changes in illumination. This can be important in scenarios where the lighting conditions may vary. Finally, in some cases, colour information might introduce unnecessary complexity, especially when dealing with noise. Converting to grayscale can help reduce the impact of noise on subsequent processing steps [5].

The line of code cv2.GaussianBlur(normal_image, (7, 7), 2) applies a Gaussian blur to the image. Gaussian blur is a blurring and smoothing operation that involves convolving the image with a Gaussian filter kernel. Gaussian blur can effectively reduce high-frequency noise in an image. High-frequency noise often appears as random variations in pixel intensity and can be smoothed out by the blurring operation. However, it is designed to preserve edges to some extent, making it a good choice when you want to reduce noise without significantly compromising edge information [4]. The parameters used in this function are:

- normal_image: The input image.
- (7, 8): The size of the Gaussian kernel. A larger kernel size results in a stronger blur effect. In this case, it's a 7x7 kernel.
- 2: The standard deviation of the Gaussian distribution. It controls the amount of smoothing applied. A higher value results in more smoothing.

After that, the images were resized. Smaller images require fewer computational resources. Resizing can speed up the processing of images, making it more efficient, especially when working with large datasets [4]. Smaller images also consume less memory, which is important for systems with limited resources. This is particularly relevant when training machine learning models or deploying applications with real-time processing constraints. In addition, smaller images can lead to faster processing times during various stages of an image processing pipeline, such as feature extraction, filtering, and defect detection.

Histogram equalization was performed to enhance the contrast of the images. Enhancing the contrast of the images can improve the visibility of defects, especially in images with uneven lighting [6].

Non-local means denoising came after. This is a technique that operates in the spatial domain and aims to reduce noise in an image by comparing and averaging similar patches of pixels. Reducing noise in the images improves the quality of subsequent processing steps.

After denoising, the Sobel filter was applied. The Sobel filter is a spatial domain filter commonly used for edge detection in image processing [4]. It is specifically designed to highlight changes in intensity, which often correspond to the boundaries of objects or structures within an image. Textile defects often manifest as changes in texture or patterns. The Sobel filter, by highlighting edges, can capture these variations and help identify regions where defects may be present. Defects in textiles can result in sudden changes in intensity, and the Sobel filter is effective at detecting such changes. It can highlight boundaries of defects that may have different textures or colours. Sobel-filtered images can serve as informative inputs for subsequent analysis steps, such as segmentation or feature extraction. It helps in preparing the image for more advanced defect detection techniques. The resulting 'sobel_filtered_image' will highlight horizontal edges in the image since it uses the horizontal kernel.

Finally, spatial domain pre-processing included the use of the Laplacian filter. The Laplacian filter is a spatial domain filter used in image processing for edge detection and enhancing regions of rapid intensity change [4]. It calculates the second derivative of the image, highlighting regions where the intensity changes abruptly. The Laplacian filter is particularly effective at emphasizing fine details and detecting edges in images. The 3x3 Laplacian filter kernel is commonly defined as follows:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Unlike the Sobel filter, which emphasizes changes in specific directions, the Laplacian filter looks for changes in intensity in all directions. This makes it suitable for detecting fine structures and details. The Laplacian filter can also be used to find zero-crossings in the filtered image, which often correspond to locations where edges are present. Zero-crossings indicate a change from dark to light or vice versa.

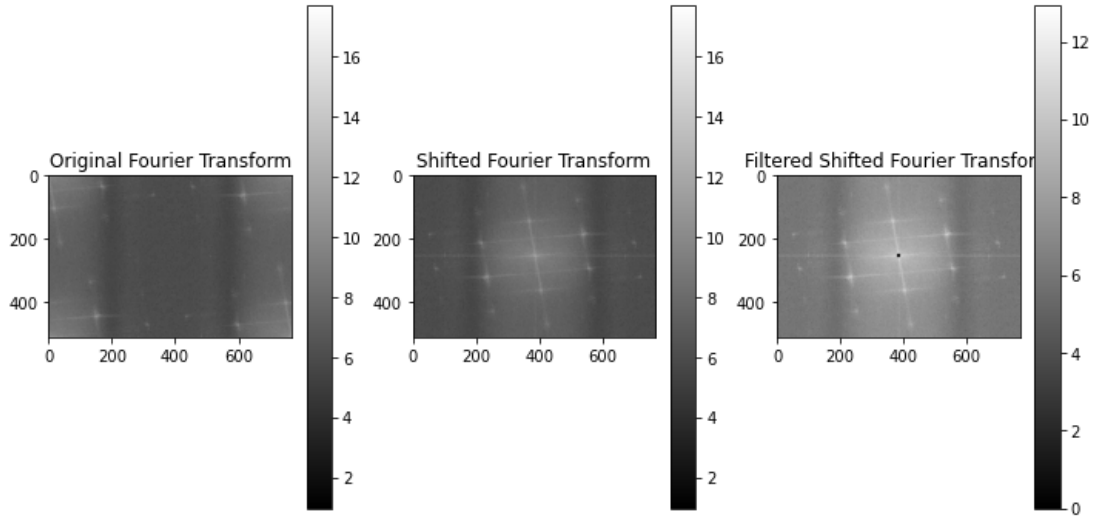
Textile defects can often be subtle and involve fine details. The Laplacian filter, with its ability to highlight rapid intensity changes, is well-suited for capturing these details. The Laplacian filter is also not directionally biased, making it suitable for detecting defects that may not align with a specific orientation. Using the Laplacian filter in conjunction with the Sobel filter can provide a more comprehensive analysis of image features, capturing both directional and nondirectional changes.

Preprocessing in the Frequency Domain

The Fourier Transform decomposes the image into its frequency components, which can be useful for analyzing and modifying specific frequency information.

The line f_transform = np.fft.fft2(image) applies the 2D Fourier Transform to the grayscale image using NumPy's fft2 function. np.fft.fft2 computes the 2-dimensional discrete Fourier Transform of the input image. The result (f_transform) is a complex array containing the frequency components of the image. f_transform_magnitude = np.abs(f_transform) calculates the magnitude spectrum of the Fourier Transform. np.abs returns the magnitude (absolute value) of the complex numbers in f_transform. f_transform_magnitude now contains the magnitude information, representing the amplitude of different frequencies in the image. f_transform_shifted = np.fft.fftshift(f_transform) performs a shift in the frequency domain to bring the zero frequency component (DC component) to the centre. np.fft.fftshift shifts the zero-frequency component of the Fourier Transform to the centre of the array. The result (f_transform_shifted) is a shifted version of the Fourier Transform. f_transform_magnitude_shifted = np.abs(f_transform_shifted)calculates the magnitude spectrum of the shifted Fourier Transform (f_transform_shifted). f_transform_magnitude_shifted now contains the magnitude information of the shifted Fourier Transform.

The diagram below depicts the original and shifted Fourier Transforms.



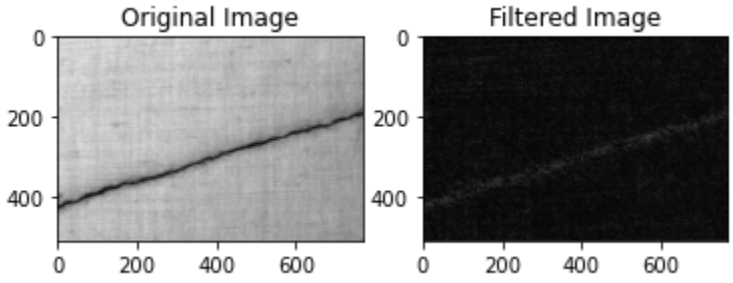
Filtering

The defects range from large-scale defects or patterns that are distributed over a broader area to fine details, small defects, and features with rapid intensity changes. The session included an experimentation with both high pass and low pass filters.

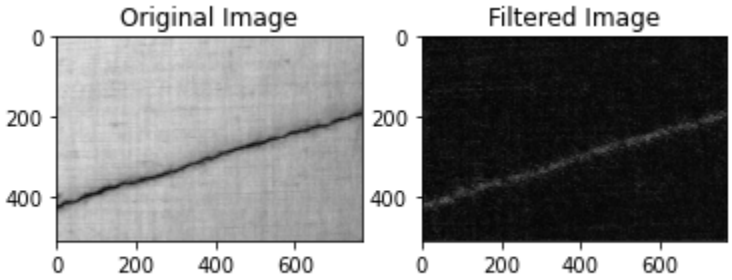
- Low pass filter: Low-pass filters allow low-frequency components (smooth changes) to pass through while attenuating high-frequency components (rapid changes). In the context of defect detection, this helps preserve overall texture and structure while reducing noise and fine details that may not be relevant to defects.

  The variable D represents the cutoff frequency of the low-pass filter. The cutoff frequency is adjusted based on the characteristics of the images. Higher values preserve more low-frequency information, while lower values emphasize high frequencies. To balance the 2, the cutoff frequency of 35Hz was selected.

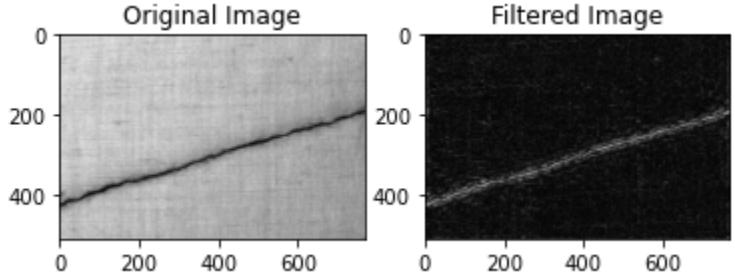
  The following are the results of experimentation with the defective image at different values of D to see which value would result in better view of the defects.



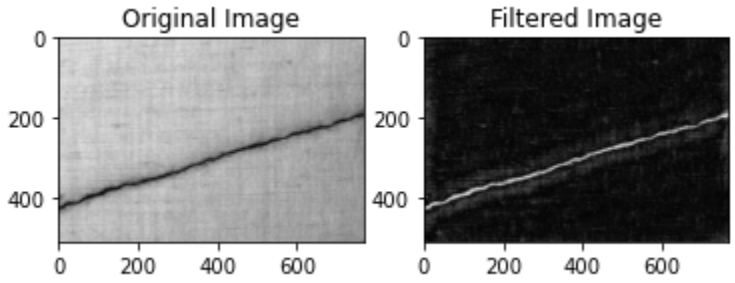
D=35



D = 25

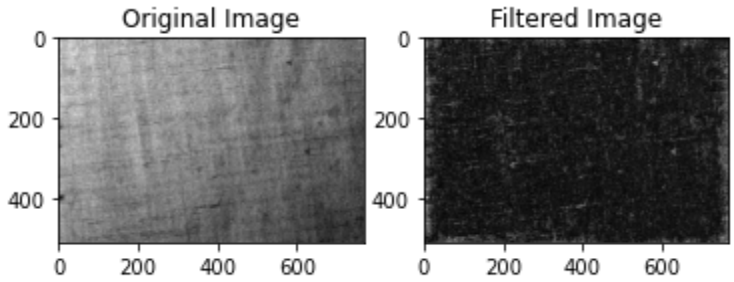


D = 15

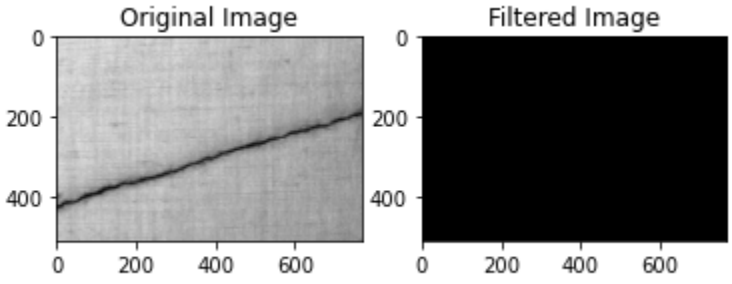


D = 5

From the above updates, it can be deduced that D=5 results in the best depiction of the defect. The diagram below shows the normal image before and after going through the low-pass filter.



- High-pass filters accentuate high-frequency components, emphasizing fine details and edges. This can be useful for detecting defects with sharp intensity changes. The images below show the differences between the original and filtered images.



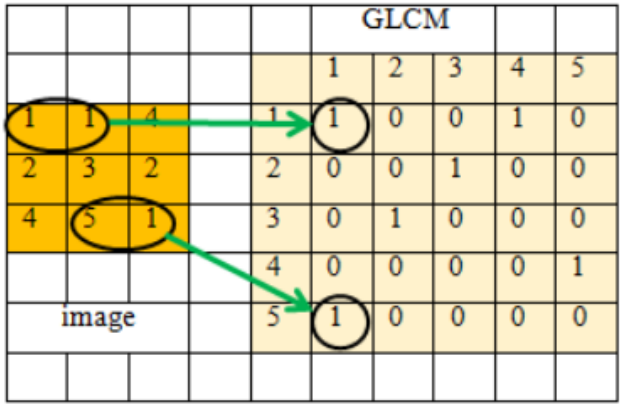
From the results, it can be deduced that high-pass filters were less than ideal for this project.

Feature Extraction

Feature extraction plays a crucial role in defect detection by transforming raw data, such as images, into a set of relevant features that can be used to characterize and distinguish between normal and defective regions. The primary goals of feature extraction in defect detection include

enhancing the discriminatory power of the data, reducing dimensionality, and providing meaningful representations for downstream processing, such as classification.

In the context of defect detection in textiles, where visual patterns and textures are essential, feature extraction methods need to capture information that reflects the unique characteristics of defects. One widely used technique for this purpose is the Gray-Level Co-occurrence Matrix (GLCM) method . The GLCM method analyzes the spatial relationships of pixel intensities in an image. It constructs a matrix that represents how often different combinations of pixel intensities occur in proximity. According to the authors [4], GLCM-based features, including contrast, energy, entropy, and homogeneity, quantify the textural properties of an image. In textile defect detection, GLCM features can help identify irregularities in patterns, variations in texture, and subtle differences that may indicate defects [1].

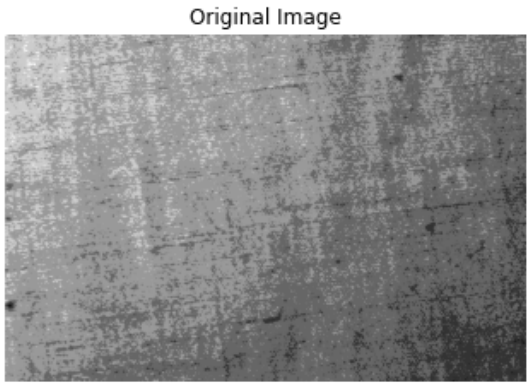


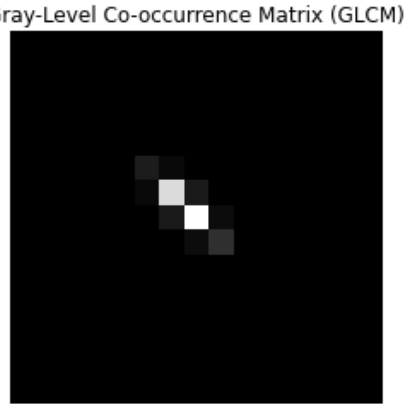
Normalization is a critical step in feature extraction to bring all features to a common scale. This is important because features may have different ranges, and normalization ensures that no single feature dominates the analysis due to its scale [2]. Common normalization techniques include Min-Max scaling, Z-score normalization, and robust normalization. In textile defect detection, normalization of GLCM features ensures that each textural property contributes proportionately to the overall defect detection process, preventing bias towards certain features and improving the robustness of the detection algorithm.

The key GLCM hyperparameters that can be tuned are levels, distances, and angles. levels determines the number of discrete levels that pixel intensities will be quantized into before computing the Gray-Level Co-occurrence Matrix (GLCM). It essentially controls the granularity of intensity values in the GLCM. Higher values provide more detail but may increase computational complexity. Distances is a list specifying the pixel distances for which the GLCM is calculated. It defines how far apart pixels should be considered in the co-occurrence computation. angles is a list specifying the angles (in degrees) at which the GLCM is calculated. It defines the directionality of the co-occurrence relationships.

The results of the 15-level GLCM are as shown below:

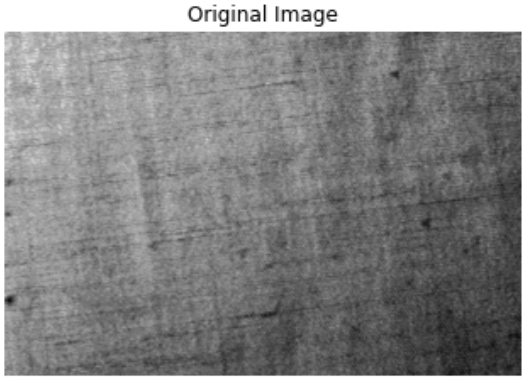
```
glcm size (15, 15, 1, 1)
Contrast: [[0.14727632]]
Dissimilarity: [[0.14727632]]
Homogeneity: [[0.92636184]]
Energy: [[0.53262902]]
Correlation: [[0.86691868]]
```

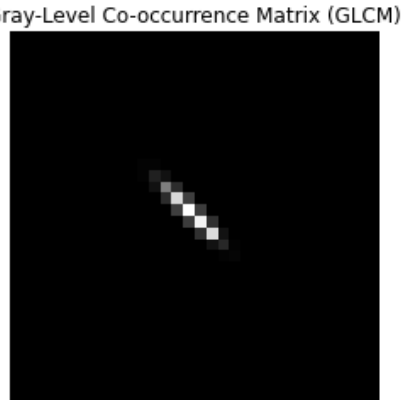




Changing the parameters resulted in the following output:’

```
glcm size (32, 32, 1, 1)
Contrast: [[0.31589696]]
Dissimilarity: [[0.31438946]]
Homogeneity: [[0.84295602]]
Energy: [[0.30828738]]
Correlation: [[0.93229176]]
```



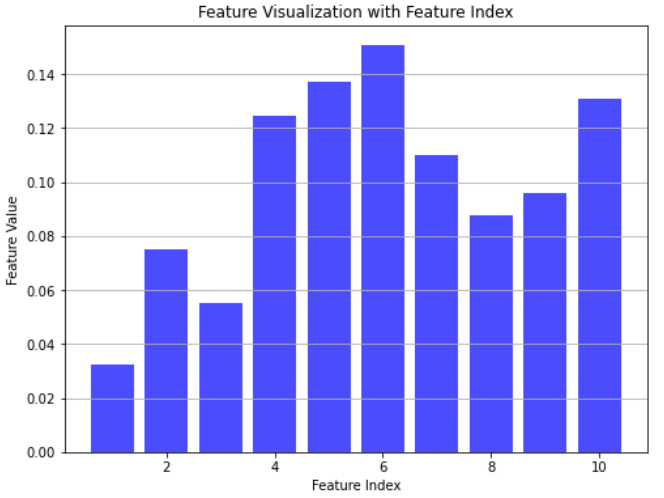


The GLCM with more levels (32), both distances (1 and 2), multiple angles, and more computation efficiency performed better than the first. It had higher contrast, dissimilarity, homogeneity, energy, and correlation. Higher contrast values often suggest that the GLCM is capturing variations in pixel intensity more effectively. Higher dissimilarity values may suggest that the GLCM is sensitive to changes in texture patterns. Higher homogeneity values may imply

that the GLCM is better at highlighting regions with similar pixel values. Finally, higher energy values may imply that the GLCM is capturing regular and consistent patterns in the image. Local Binary Pattern (LBP) is a texture descriptor used in image analysis and computer vision. It quantifies the local patterns in an image by comparing the intensity of each pixel with its neighbouring pixels. For each pixel in the image, LBP compares its intensity with the intensities of its surrounding pixels. The result of each comparison is encoded as a binary digit (0 or 1). The binary digits are then concatenated to form a binary pattern for that particular pixel. The histogram of LBP values provides a condensed and normalized representation of the distribution of local patterns in an image. It summarizes the occurrence frequencies of different LBP patterns. The histogram serves as a feature vector that can be used for further analysis and classification tasks.

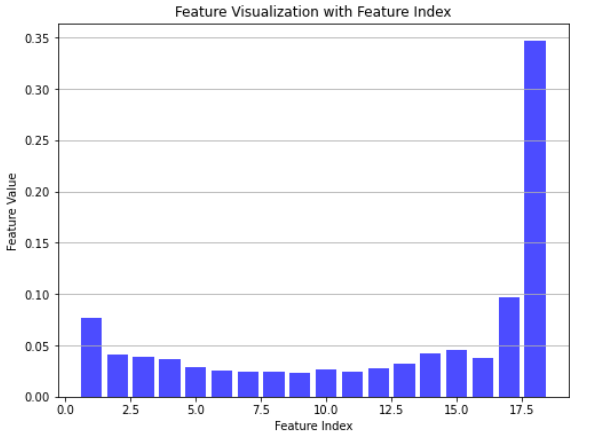
The diagram below shows the LBP output and histogram for a radius equal to 1. The histogram is characterized by fewer bins, suggesting a coarser representation of LBP patterns. Lower variability may indicate a broader analysis of texture features.

```
LBP Histogram:
[0.03255208 0.07517751 0.05520884 0.12435913 0.13733927 0.15068054
 0.10988871 0.08794149 0.09616089 0.13069153]
```

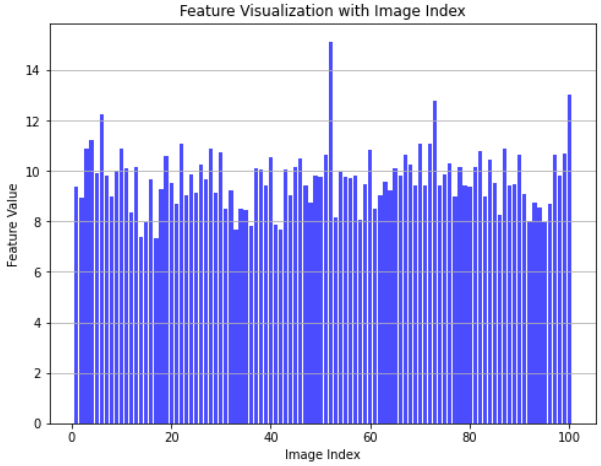


After adjusting the radius to 2, the folowing results were found. The histogram reflects a more detailed and diverse distribution of LBP patterns. Higher variability in pattern occurrences indicates a finer analysis of texture details.

```
LBP Histogram:
[0.07718404 0.04130554 0.0387853  0.0366567  0.02828217 0.02579753
 0.0238088  0.02386729 0.02360535 0.0266215  0.02466075 0.02789052
 0.03215281 0.04216258 0.04563904 0.03804016 0.09692891 0.34661102]
```



The diagram below shows a histogram of a feature selected from the array of images index.



Model Building

Research by [3] and [5] demonstrated a machine learning approach utilizing SVM and KNN methods performed incredibly well in detecting diverse fabric defects. KNN has a parameter (n_neighbors) and SVM has three parameters (C, kernel, and gamma) that can be modified to improve outputs and accuracy levels of the detection process.

Using n_neighbors equal to 2 and n_neighbors equal to 4 resulted in lower accuracies of 85% for both of them.

```
# Create a KNN classifier
knn_classifier = KNeighborsClassifier(n_neighbors=2)  # You can adjust the number of neighbors (k) as needed

# Train the KNN classifier
knn_classifier.fit(X_train, y_train)

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

# Calculate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

```
Accuracy: 0.85
```

```
# Create a KNN classifier
knn_classifier = KNeighborsClassifier(n_neighbors=4)  # You can adjust the number of neighbors (k) as needed

# Train the KNN classifier
knn_classifier.fit(X_train, y_train)

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

# Calculate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

```
Accuracy: 0.85
```

It can therefore be deduced that the n_neighbors value equal to 3 is optimum for higher accuracy as shown below.

```
# Create a KNN classifier
knn_classifier = KNeighborsClassifier(n_neighbors=3)  # From experiementation, 3 is optimum for accuarcy

# Train the KNN classifier
knn_classifier.fit(X_train, y_train)

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

# Calculate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

```
Accuracy: 0.90
```

The SVM classifier with C=1.0, kernel='linear', gamma=scale resulted in an accuracy of 90% as shwon below. C is the regularization parameter that controls model complexity (higher C means stricter separation, lower C allows more flexibility). The kernel parameter specifies the kernel function for non-linear classification ('linear', 'rbf', 'poly', 'sigmoid'). Finally, gamma is the kerrnel coefficient (for 'rbf', 'poly', 'sigmoid') that influences decision boundary shape.

```
# Create a Support Vector Machine (SVM) classifier
svm_classifier = SVC(C=1.0, kernel='linear', gamma='scale')

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

```
SVC(kernel='linear')
```

```
# Make predictions on the test data
y_pred = svm_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)

# Print the accuracy of the classifier
print(f"Accuracy of the SVM classifier: {accuracy:.2f}")
```

```
Accuracy of the SVM classifier: 0.90
```

Changing the values to C=0.1, kernel='rbf', gamma=0.001 resulted in a drop of accuracy to 40%.

```
# Create a Support Vector Machine (SVM) classifier
svm_classifier = SVC(C=0.1, kernel='rbf', gamma=0.001)

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

```
SVC(C=0.1, gamma=0.001)
```

```
# Make predictions on the test data
y_pred = svm_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)

# Print the accuracy of the classifier
print(f"Accuracy of the SVM classifier: {accuracy:.2f}")
```

```
Accuracy of the SVM classifier: 0.40
```

```
# Create a Support Vector Machine (SVM) classifier
svm_classifier = SVC(C=100, kernel='rbf', gamma=0.001)

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

```
SVC(C=100, gamma=0.001)
```

```
# Make predictions on the test data
y_pred = svm_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)

# Print the accuracy of the classifier
print(f"Accuracy of the SVM classifier: {accuracy:.2f}")
```

```
Accuracy of the SVM classifier: 0.85
```

```
# Create a Support Vector Machine (SVM) classifier
svm_classifier = SVC(C=10, kernel='linear', gamma=0.1)

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

```
SVC(C=10, gamma=0.1, kernel='linear')
```

```

# Make predictions on the test data
y_pred = svm_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)

# Print the accuracy of the classifier
print(f"Accuracy of the SVM classifier: {accuracy:.2f}")
```

```
Accuracy of the SVM classifier: 0.90
```

Using GridSearchCV generated the following results for prime hyperparameters.

```
from sklearn.model_selection import GridSearchCV

# Parameter grid for SVM
param_grid = {'C': [0.1, 1, 10, 100], 'kernel': ['linear', 'rbf', 'poly'], 'gamma': ['scale', 0.001, 0.01, 0.1]}

# Create GridSearchCV object
grid_search = GridSearchCV(svm_classifier, param_grid, cv=5)  # cv=5 for 5-fold cross-validation

# Perform grid search
grid_search.fit(X_train, y_train)

# Get best parameters and model
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_
print(f"Best parameters: {best_params}")
print(f"Best model: {best_model}")

# Print best accuracy
print(f"Best accuracy: {grid_search.best_score_:.2f}")
```

```
Best parameters: {'C': 0.1, 'gamma': 'scale', 'kernel': 'linear'}
Best model: SVC(C=0.1, kernel='linear')
Best accuracy: 0.94
```

Using parameters provided by GridSearchCV resulted in the following:

```
# Create a Support Vector Machine (SVM) classifier
svm_classifier = SVC(C=0.1, kernel='linear', gamma='scale')

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

```
SVC(C=0.1, kernel='linear')
```

```

# Make predictions on the test data
y_pred = svm_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)

# Print the accuracy of the classifier
print(f"Accuracy of the SVM classifier: {accuracy:.2f}")
```

```
Accuracy of the SVM classifier: 0.95
```

```
# Create a Support Vector Machine (SVM) classifier
svm_classifier = SVC(C=0.1, kernel='linear', gamma=0.1)

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

```
SVC(C=0.1, gamma=0.1, kernel='linear')
```

```
# Make predictions on the test data
y_pred = svm_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)

# Print the accuracy of the classifier
print(f"Accuracy of the SVM classifier: {accuracy:.2f}")
```

```
Accuracy of the SVM classifier: 0.95
```

After experimenting with parameters, C=0.1, kernel = 'linear' and gamma = 0.1 or gamma = 'scale' were found to give the highest accuracy of 95%.

Conclusion

In this laboratory session, I embarked on a comprehensive exploration of defect detection in textiles, employing a variety of image processing techniques and machine learning methodologies. Converting the image to grayscale, using Gaussian blur, histogram equalization, non-local means denoising, and applying a Sobel filter were the most useful preprocessing techniques in the spatial domain for this project. In addition, in the frequency domain, a low-pass filter with the cutoff frequency of 5Hz was found to give the best results and perform better than high-pass filters. After experimenting with a variety of preprocessing methods, I concluded that changing preprocessing techniques changes the appearance of the images and the accuracy of the resulting models. Feature extraction emerged as a critical component in the defect detection pipeline. Leveraging methods like GLCM (Grey Level Co-occurrence Matrix) and LBP (Local Binary Pattern), I harnessed the power of texture analysis to quantify statistical and textural properties. The extracted texture features, including contrast, dissimilarity, and entropy were more instrumental in characterizing distinguishable defect patterns compared to statistical features like the mean, variance, and standard deviation of the intensities. Considering machine learning models, SVM (Support Vector Machine) and KNN were employed as robust classifiers to discern between normal and defective textiles. By tuning the parameters, SVM produced a higher accuracy of 95% compared to KNN. I therefore concluded that different parameters resulted in varying accuracies of classification models.

References

[1] J. Jing, H. Ma, and H. Zhang, “Automatic fabric defect detection using a deep convolutional neural network,” *Coloration Technology*, vol. 135, no. 3, pp. 213–223, 2019. doi:10.1111/cote.12394

[2] V. Raut and I. Singh, “Digital image processing based automatic fabric defect detection techniques: A survey,” *Lecture Notes in Electrical Engineering*, pp. 1029–1038, 2020. doi:10.1007/978-981-15-7031-5_98

[3] A. Rasheed *et al.*, “Fabric defect detection using Computer Vision Techniques: A comprehensive review,” *Mathematical Problems in Engineering*, vol. 2020, pp. 1–24, 2020. doi:10.1155/2020/8189403

[4] S. Priya, T. Ashok Kumar and V. Paul, "A novel approach to fabric defect detection using digital image processing," 2011 International Conference on Signal Processing, Communication, Computing and Networking Technologies, Thuckalay, India, 2011, pp. 228-232, doi: 10.1109/ICSCCN.2011.6024549.

[5]H. Y. Riskiawan, T. D. Puspitasari, F. I. Hasanah, N. D. Wahyono and M. F. Kurnianto, "Identifying Cocoa ripeness using K-Nearest Neighbor (KNN) Method," 2018 International Conference on Applied Science and Technology (iCAST), Manado, Indonesia, 2018, pp. 354-357, doi: 10.1109/iCAST1.2018.8751633.

[6] K. Rahimunnisa, “Textile fabric defect detection,” *Journal of Innovative Image Processing*, vol. 4, no. 3, pp. 165–172, 2022. doi:10.36548/jiip.2022.3.004.