```python
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report,confusion_matrix
import tensorflow as tf
import keras
import os
import cv2
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import applications
from keras.models import Sequential, load_model
from keras.preprocessing import image
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D,
Flatten, Dense, Dropout
from mtcnn import MTCNN
from skimage.feature import hog
from skimage import exposure
from skimage.feature import local_binary_pattern
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

df = pd.read_csv('faces.csv')
df.head()
```

	image_name	width	height	x0	y0	x1	y1
0	00001722.jpg	1333	2000	490	320	687	664
1	00001044.jpg	2000	1333	791	119	1200	436
2	00001050.jpg	667	1000	304	155	407	331
3	00001736.jpg	626	417	147	14	519	303
4	00003121.jpg	626	418	462	60	599	166

Step 1: Data Preprocessing

```python
# # Remove duplicate or irrelevant images and correct any incorrect
annotations.

# # Inspect for duplicates
duplicates = df[df.duplicated(subset='image_name', keep=False)]
print("Duplicate Images:\n", duplicates)
```

Duplicate Images:

	image_name	width	height	x0	y0	x1	y1
4	00003121.jpg	626	418	462	60	599	166
5	00003121.jpg	626	418	316	157	441	254
6	00003121.jpg	626	418	35	71	160	168
7	00003121.jpg	626	418	254	94	376	187
8	00003121.jpg	626	418	166	118	306	226

```
...             ...     ...     ...   ...   ...   ...   ...
3345  00002232.jpg     620     349     4    36   186   158
3346  00002232.jpg     620     349   122   103   344   248
3347  00002232.jpg     620     349   258   118   541   303
3348  00002232.jpg     620     349   215    11   362   108
3349  00002232.jpg     620     349   330     1   487    81

[1637 rows x 7 columns]
```

Summary:

The dataset contains multiple entries with the same image_name but different bounding box coordinates. This indicates the presence of multiple faces detected within the same image. These duplicate entries should not be removed, as they provide valuable information about multiple face annotations per image.

```
# # 1. Remove duplicates
# df = df.drop_duplicates(subset='image_name').reset_index(drop=True)

df.info
```

```
<bound method DataFrame.info of           image_name  width  height   x0
y0    x1   y1
0     00001722.jpg   1333    2000   490   320   687   664
1     00001044.jpg   2000    1333   791   119  1200   436
2     00001050.jpg    667    1000   304   155   407   331
3     00001736.jpg    626     417   147    14   519   303
4     00003121.jpg    626     418   462    60   599   166
...            ...    ...     ...   ...   ...   ...   ...
3345  00002232.jpg    620     349     4    36   186   158
3346  00002232.jpg    620     349   122   103   344   248
3347  00002232.jpg    620     349   258   118   541   303
3348  00002232.jpg    620     349   215    11   362   108
3349  00002232.jpg    620     349   330     1   487    81

[3350 rows x 7 columns]>
```

```python
images_path = 'C:/Users/DELL/Desktop/AIML/accessments/images-20241204T113424Z-001/images/'
# Ensure that all images in the DataFrame exist in the directory
def validate_images(df, images_path):
    valid_rows = []
    for _, row in df.iterrows():
        image_path = os.path.join(images_path, row['image_name'])
        if os.path.exists(image_path):
            valid_rows.append(row)
    return pd.DataFrame(valid_rows)

df = validate_images(df, images_path)
print(f"Valid images count: {len(df)}")
```

```
Valid images count: 3350
```

Validation of the dataset against the images folder has been successfully completed. All 3,350 images referenced in the dataset are confirmed to exist in the specified directory. This ensures the dataset integrity and readiness for further processing and analysis.

```python
# Check if bounding box coordinates are logical
def validate_annotations(row):
    if row['x0'] >= row['x1'] or row['y0'] >= row['y1']:
        return False

    # Check if bounding box is within image dimensions
    if row['x0'] < 0 or row['y0'] < 0 or row['x1'] > row['width'] or row['y1'] > row['height']:
        return False

    # Optional: Check bounding box area (if needed)
    box_area = (row['x1'] - row['x0']) * (row['y1'] - row['y0'])
    if box_area <= 0:  # Example threshold: area must be positive
        return False
    return True

# Apply validation to each row in the DataFrame
df['is_valid'] = df.apply(validate_annotations, axis=1)
df.head()
```

	image_name	width	height	x0	y0	x1	y1	is_valid
0	00001722.jpg	1333	2000	490	320	687	664	True
1	00001044.jpg	2000	1333	791	119	1200	436	True
2	00001050.jpg	667	1000	304	155	407	331	True
3	00001736.jpg	626	417	147	14	519	303	True
4	00003121.jpg	626	418	462	60	599	166	True

```python
# Filter rows with is_valid set to False
invalid_bounding_boxes = df[df['is_valid'] == False]

# Check if there are any invalid rows
if len(invalid_bounding_boxes) > 0:
    print(f"Found {len(invalid_bounding_boxes)} invalid bounding box annotations:")
    print(invalid_bounding_boxes)
else:
    print("All bounding box annotations are valid!")
```

```
All bounding box annotations are valid!
```

Summary of Bounding Box Validation

Validation Goal: Checked the logical correctness of bounding box annotations, ensuring:

x0 < x1 and y0 < y1 (coordinates are logically aligned).

Bounding boxes are within the image dimensions (x0, y0 ≥ 0 and x1 ≤ width, y1 ≤ height).

The bounding box area is positive.

This confirms that all annotations in the dataset are correctly formatted and logically valid. You can proceed with further analysis or model training without concerns about annotation errors.

```python
# Calculating the box area and add a new column having the bocx area
def box_area(row):
    box_area = (row['x1'] - row['x0']) * (row['y1'] - row['y0'])
    return box_area

# Apply validation to each row in the DataFrame
df['box_area'] = df.apply(box_area, axis=1)
df.head()
```

```
     image_name  width  height   x0   y0    x1   y1  is_valid
box_area
0  00001722.jpg   1333    2000  490  320   687  664      True
67768
1  00001044.jpg   2000    1333  791  119  1200  436      True
129653
2  00001050.jpg    667    1000  304  155   407  331      True
18128
3  00001736.jpg    626     417  147   14   519  303      True
107508
4  00003121.jpg    626     418  462   60   599  166      True
14522
```

Step 2: Exploratory Data Analysis (EDA)

```python
# Image Count
# Total number of images
total_images_unique = df['image_name'].nunique()
total_images = df['image_name'].count()
print(f"Total number of unique images: {total_images_unique}")
print(f"Total number of images: {total_images}")
```

```
Total number of unique images: 2204
Total number of images: 3350
```

```python
# Face Count
df['face_count'] = df.apply(lambda row: 1, axis=1)  # Each row
represents one face
total_faces = df['face_count'].sum()
print(f"Total number of faces in the dataset: {total_faces}")
```

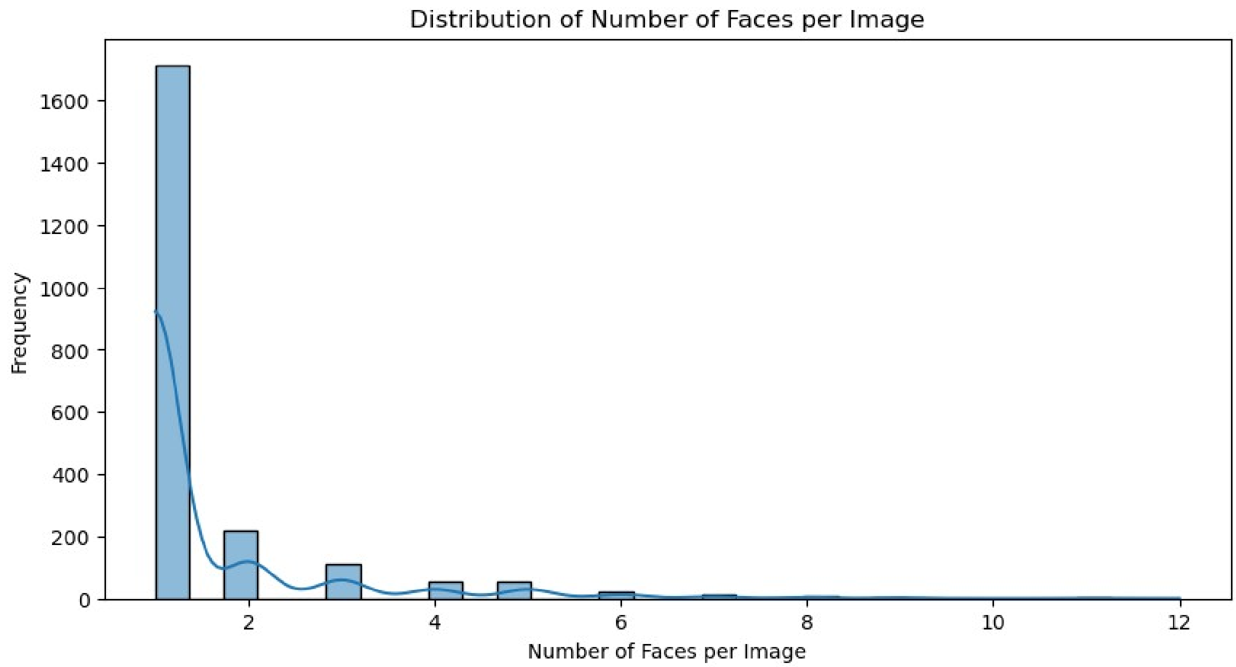
```
Total number of faces in the dataset: 3350
```

Summary of Face Count Calculation

Assigned a face_count value of 1 for each row in the dataset, as each row corresponds to a unique face.

```python
# Number of Faces per Image
image_face_counts = df.groupby('image_name')['face_count'].count()
plt.figure(figsize=(10, 5))
```

```
sns.histplot(image_face_counts, bins=30, kde=True)
plt.title('Distribution of Number of Faces per Image')
plt.xlabel('Number of Faces per Image')
plt.ylabel('Frequency')
plt.show()
```



Approach:

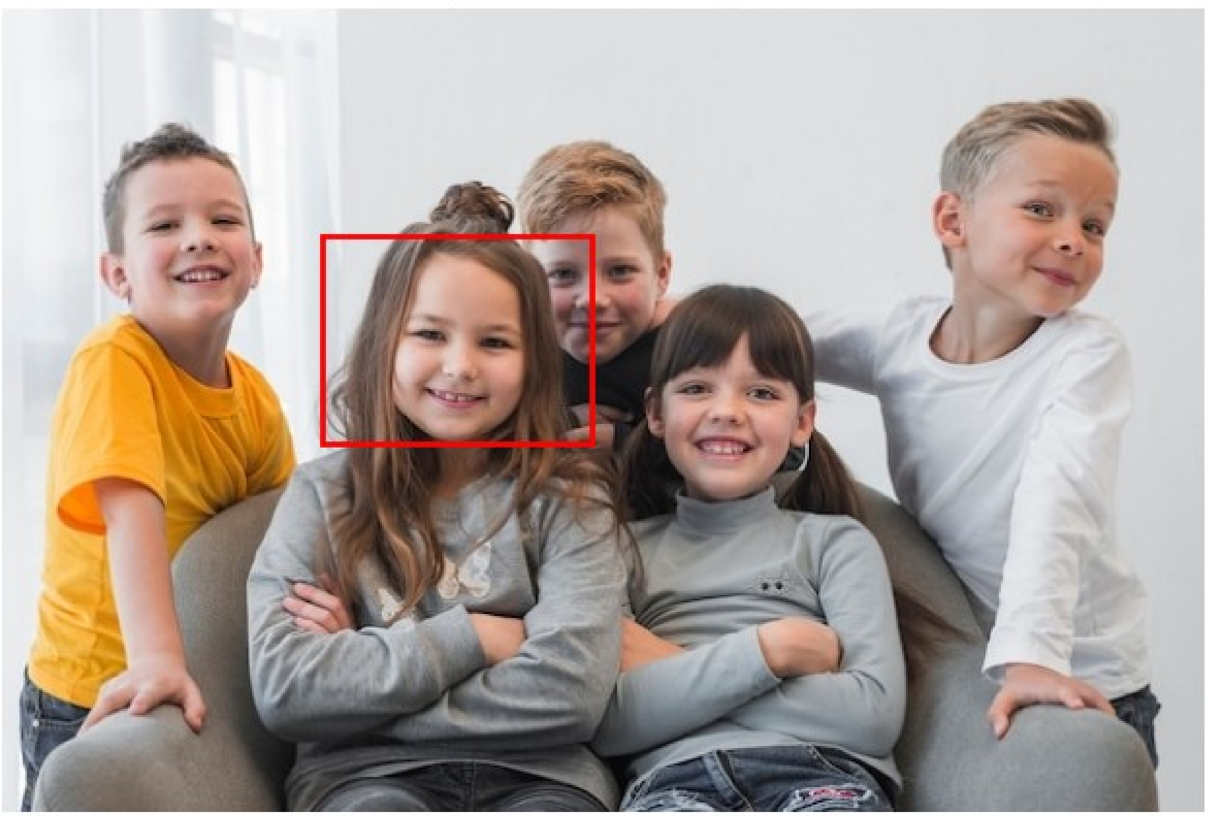
Grouped the dataset by image_name and counted the number of rows (faces) associated with each image.

Generated a histogram with a KDE (Kernel Density Estimate) plot to show the distribution of face counts per image.

```
# Bounding Box Accuracy (Visualization)
def plot_image_with_bounding_box(image_path, x0, y0, x1, y1):
    image = cv2.imread(image_path)
```

```
    if image is None:
        return None
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    plt.figure(figsize=(8, 8))
    plt.imshow(image)
    rect = plt.Rectangle((x0, y0), x1 - x0, y1 - y0, linewidth=2,
edgecolor='red', facecolor='none')
    plt.gca().add_patch(rect)
    plt.title('Image with Bounding Box')
    plt.axis('off')
    plt.show()

# Display a sample image with its bounding box
sample_row = df.iloc[8]  # Change index to pick a different sample
image_path = os.path.join(images_path, sample_row['image_name'])
plot_image_with_bounding_box(image_path, sample_row['x0'],
sample_row['y0'], sample_row['x1'], sample_row['y1'])
```

Image with Bounding Box



```
# Label Consistency
# Check for duplicate annotations for the same image
duplicate_annotations = df[df.duplicated(subset=['image_name', 'x0',
'y0', 'x1', 'y1'], keep=False)]
```

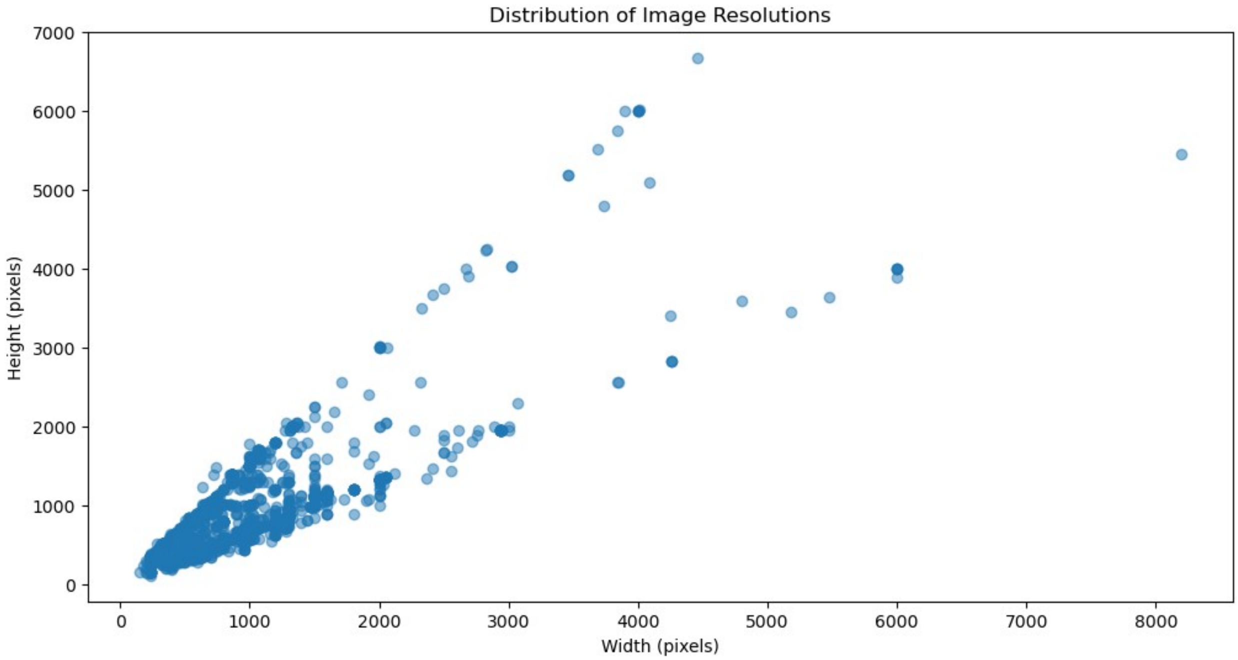
```
print(f"Number of duplicate bounding box annotations:
{len(duplicate_annotations)}")
if len(duplicate_annotations) > 0:
    print("Sample duplicate annotations:")
    print(duplicate_annotations.head())
else:
    print("No duplicates")
```

```
Number of duplicate bounding box annotations: 0
No duplicates
```

No duplicate bounding box annotations were found in the dataset.

This indicates that the annotations are unique and consistent for each face detected in the images.

```
# Step 2.6: Resize Requirements
image_sizes = []
for image_name in df['image_name'].unique():
    image_path = os.path.join(images_path, image_name)
    if os.path.exists(image_path):
        img = cv2.imread(image_path)
        if img is not None:
            h, w, _ = img.shape
            image_sizes.append((w, h))

image_sizes = np.array(image_sizes)
plt.figure(figsize=(12, 6))
plt.scatter(image_sizes[:, 0], image_sizes[:, 1], alpha=0.5)
plt.title('Distribution of Image Resolutions')
plt.xlabel('Width (pixels)')
plt.ylabel('Height (pixels)')
plt.show()
```


Distribution of Image Resolutions
Height (pixels)
Width (pixels)
0
1000
2000
3000
4000
5000
6000
7000
8000

The variation in image resolutions indicates a lack of uniformity, with some images being significantly larger or smaller than others.

Resizing Required: Resizing the images to a standard resolution (e.g., 224*224 or 331*331 pixels) is necessary to ensure compatibility with the model architecture.

A standard resizing approach should maintain the aspect ratio to avoid distortion while balancing memory usage and model performance.

```python
# Check if images have clear resolutions suitable for detection (e.g., > 300x300)
resolution_filter = (image_sizes[:, 0] == 331) & (image_sizes[:, 1] == 331)
valid_resolutions = np.sum(resolution_filter)
print(f"Number of images with suitable resolution (331*331: {valid_resolutions} out of {len(df)}")
```

```
Number of images with suitable resolution (331*331: 0 out of 3350
```

```python
# Get image resolution from the DataFrame
df['resolution'] = df.apply(lambda row: (row['width'], row['height']), axis=1)

df.head()
```

```
     image_name  width  height   x0   y0    x1   y1  is_valid  box_area  \
0  00001722.jpg   1333    2000  490  320   687  664      True     67768
1  00001044.jpg   2000    1333  791  119  1200  436      True    129653
2  00001050.jpg    667    1000  304  155   407  331      True     18128
```

```
3  00001736.jpg    626     417  147   14   519  303      True
107508
4  00003121.jpg    626     418  462   60   599  166      True
14522

     resolution
0  (1333, 2000)
1  (2000, 1333)
2   (667, 1000)
3    (626, 417)
4    (626, 418)
```

Step 3: Feature Engineering

```
# Bounding Box Coordinates: Use the coordinates of bounding boxes to
define the location of faces in images.
# Display a sample image with its bounding box
sample_row = df.iloc[15]  # Change index to pick a different sample
image_path = os.path.join(images_path, sample_row['image_name'])
plot_image_with_bounding_box(image_path, sample_row['x0'],
sample_row['y0'], sample_row['x1'], sample_row['y1'])
```

Image with Bounding Box


```
df.head()

     image_name  width  height   x0   y0    x1   y1  is_valid
box_area  \
0  00001722.jpg   1333    2000  490  320   687  664      True
67768
1  00001044.jpg   2000    1333  791  119  1200  436      True
129653
2  00001050.jpg    667    1000  304  155   407  331      True
18128
3  00001736.jpg    626     417  147   14   519  303      True
107508
4  00003121.jpg    626     418  462   60   599  166      True
14522

     resolution
0  (1333, 2000)
1  (2000, 1333)
2   (667, 1000)
3    (626, 417)
4    (626, 418)
```

```python
# Face Landmarks: Extract facial landmarks (e.g., eyes, nose, mouth)
for more detailed face detection.
# Load the MTCNN detector
detector = MTCNN()

# Load an image
image = cv2.imread(image_path)  # Using existing image only for sample
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Convert to RGB
for MTCNN

# Detect faces and landmarks
results = detector.detect_faces(image_rgb)

results
```

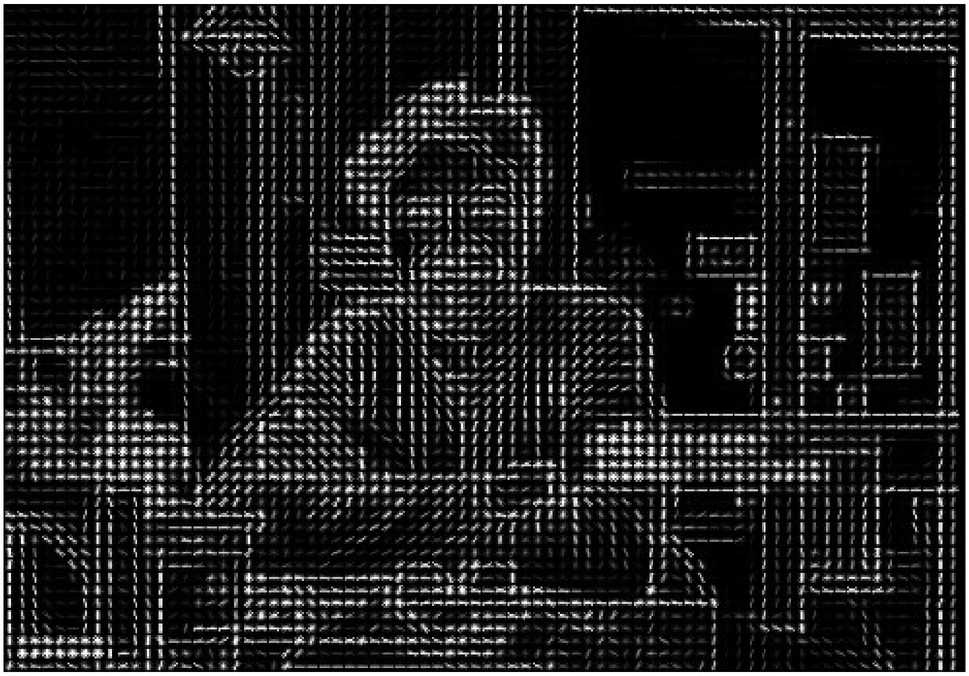
```
[{'box': [245, 86, 84, 113],
  'confidence': 0.9991952776908875,
  'keypoints': {'nose': [286, 157],
   'mouth_right': [309, 167],
   'right_eye': [306, 129],
   'left_eye': [266, 132],
   'mouth_left': [269, 172]}}]
```

```python
# Plot the image with landmarks
# Create the plot
fig, ax = plt.subplots(figsize=(8, 6))  # Create a figure and axes
ax.imshow(image_rgb)  # Display the image

# Overlay bounding boxes and landmarks
for result in results:
    # Draw bounding box
    bounding_box = result['box']  # [x, y, width, height]
    x, y, width, height = bounding_box
    rect = plt.Rectangle((x, y), width, height, linewidth=2,
edgecolor='red', facecolor='none')
    ax.add_patch(rect)  # Add rectangle to axes

    # Plot landmarks
    landmarks = result['keypoints']  # Get facial landmarks
    for key, point in landmarks.items():  # e.g., 'left_eye',
'right_eye', etc.
        ax.scatter(point[0], point[1], s=40, c='blue', marker='o')  #
Add landmarks

ax.axis('off')  # Turn off axis for a cleaner display
plt.show()  # Render the figure
```

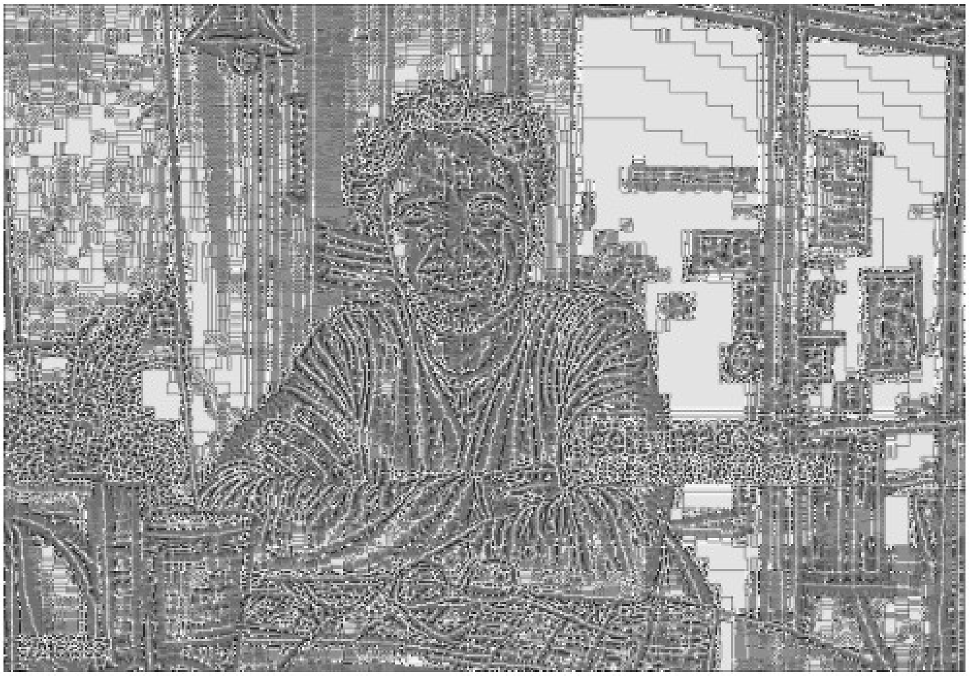

```
# 3.3 Histogram Equalization: Enhance image contrast to improve face
visibility.
def histogram_equalization(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    equalized = cv2.equalizeHist(gray)
    return equalized

# Test histogram equalization
equalized_image = histogram_equalization(image)
plt.imshow(equalized_image, cmap='gray')
plt.axis('off')
plt.show()
```


```
# 3.4 Normalization
def normalize_image(image):
    return image / 255.0

normalized_image = normalize_image(image)
plt.imshow(normalized_image)
plt.axis('off')
plt.show()
```


```python
# 3.5 HOG (Histogram of Oriented Gradients)
# Function to extract HOG features
def extract_hog_features(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    features, hog_image = hog(gray, pixels_per_cell=(8, 8),
cells_per_block=(2, 2), visualize=True, block_norm='L2-Hys')
    hog_image_rescaled = exposure.rescale_intensity(hog_image,
in_range=(0, 10))
    return features, hog_image_rescaled

# Test HOG feature extraction
features, hog_image = extract_hog_features(image)
plt.imshow(hog_image, cmap='gray')
plt.axis('off')
plt.show()
```



```
# 3.6 LBP (Local Binary Patterns)
# Function to extract LBP features
def extract_lbp_features(image, radius=1, points=8):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    lbp_image = local_binary_pattern(gray, points, radius,
method='uniform')
    return lbp_image

# Test LBP feature extraction
lbp_image = extract_lbp_features(image)
plt.imshow(lbp_image, cmap='gray')
plt.axis('off')
plt.show()
```



Step 4: Preprocessing Images

```
# Function to evaluate resolution and resize if necessary
def evaluate_resolution(image, min_resolution=(224, 224)):
    if image is not None:
        image = cv2.resize(image, min_resolution)
    return image

# Function to histogram_equalization
def histogram_equalization(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    equalized = cv2.equalizeHist(gray)
    return equalized

# Function to normalize_image
def normalize_image(image):
    return image / 255.0


# Function to preprocess the image (all steps combined)
def preprocess_image(image, target_size=(224, 224),
min_resolution=(224, 224)):
    """
    Perform all preprocessing steps.
    1. Evaluate resolution and resize if necessary.
    2. Apply histogram equalization.
    3. Normalize pixel values.
```

```
    """
    image = evaluate_resolution(image, min_resolution)
    image = histogram_equalization(image)
    image = normalize_image(image)

    return image

# Initialize MTCNN detector
detector = MTCNN()

# Function to check if an image contains a face
def detect_face(image_path, resize_dims=(224, 224)):
    # Read image
    image = cv2.imread(image_path)

    if image is None:
        print(f"Error: Image {image_path} not found.")
        return 0  # No faces detected if the image is invalid

    # Resize the image to reduce memory usage
    image = cv2.resize(image, resize_dims)  # Resize for memory
optimization

    # Convert image to RGB (MTCNN expects RGB images)
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Detect faces using MTCNN
    faces = detector.detect_faces(image_rgb)

    # Return 1 if faces are detected, 0 if no face is detected
    return 1 if len(faces) > 0 else 0

# Initialize an empty list for labels
labels = []

# Batch processing loop (assuming you want to process in batches of
100)
batch_size = 100
for i, image_name in enumerate(df['image_name']):
    image_path = f"C:/Users/DELL/Desktop/AIML/accessments/images-
20241204T113424Z-001/images/{image_name}"
    label = int(detect_face(image_path))  # Convert to integer
    labels.append(label)

    # Every batch_size iterations, add labels to the DataFrame
    if (i + 1) % batch_size == 0 or (i + 1) == len(df):
        # Update the DataFrame with the current batch of labels

        df.loc[i - (len(labels) - 1):i, 'label'] = labels[-
len(df.loc[i - (len(labels) - 1):i]) :]
```

```
        labels = []

# Check if all labels have been assigned
df['label'].head()  # Ensure the labels are added correctly
```

```
0    1.0
1    1.0
2    1.0
3    1.0
4    1.0
Name: label, dtype: float64
```

```
df.head()
```

```
     image_name  width  height   x0   y0    x1   y1  is_valid  
box_area  \
0  00001722.jpg   1333    2000  490  320   687  664      True  
67768
1  00001044.jpg   2000    1333  791  119  1200  436      True  
129653
2  00001050.jpg    667    1000  304  155   407  331      True  
18128
3  00001736.jpg    626     417  147   14   519  303      True  
107508
4  00003121.jpg    626     418  462   60   599  166      True  
14522

     resolution  label
0  (1333, 2000)    1.0
1  (2000, 1333)    1.0
2   (667, 1000)    1.0
3    (626, 417)    1.0
4    (626, 418)    1.0
```

Step 6: Train the Model

```
# Prepare a list of preprocessed images to feed into the model for
training
images = []
for image_name in df['image_name']:
    # Read the image using the pre-defined image path
    image_path = f"C:/Users/DELL/Desktop/AIML/accessments/images-
20241204T113424Z-001/images/{image_name}"
    image = cv2.imread(image_path)

    # Use the existing preprocess_image function to preprocess the
image
    image = preprocess_image(image)
```

```python
    # Append the preprocessed image to the list
    images.append(image)

# Convert the list of images into a numpy array
images = np.array(images)

images
```

```
array([[[0.17254902, 0.17647059, 0.17647059, ..., 0.17254902,
         0.17254902, 0.17254902],
        [0.17254902, 0.17647059, 0.17647059, ..., 0.17254902,
         0.17254902, 0.17254902],
        [0.17647059, 0.17647059, 0.17647059, ..., 0.17254902,
         0.17254902, 0.17647059],
        ...,
        [0.94509804, 0.94509804, 0.94509804, ..., 0.94509804,
         0.94509804, 0.97254902],
        [0.94509804, 0.94509804, 0.94509804, ..., 0.94509804,
         0.94509804, 0.94509804],
        [0.94509804, 0.94509804, 0.94509804, ..., 0.94509804,
         0.94509804, 0.94509804]],

       [[0.80784314, 0.80784314, 0.8       , ..., 0.45882353,
         0.47843137, 0.47843137],
        [0.80784314, 0.80784314, 0.80784314, ..., 0.45882353,
         0.45882353, 0.49411765],
        [0.80784314, 0.80784314, 0.80784314, ..., 0.45882353,
         0.45882353, 0.47843137],
        ...,
        [0.05490196, 0.05490196, 0.05490196, ..., 0.00392157,
         0.00392157, 0.00392157],
        [0.05490196, 0.05490196, 0.05490196, ..., 0.00392157,
         0.00392157, 0.00392157],
        [0.04313725, 0.04313725, 0.04313725, ..., 0.00392157,
         0.00392157, 0.00392157]],

       [[0.        , 0.        , 0.        , ..., 0.        ,
         0.        , 0.        ],
        [0.        , 0.        , 0.        , ..., 0.        ,
         0.        , 0.        ],
        [0.        , 0.        , 0.        , ..., 0.        ,
         0.        , 0.        ],
        ...,
        [0.10196078, 0.10196078, 0.10196078, ..., 0.16470588,
         0.13333333, 0.13333333],
        [0.10196078, 0.10196078, 0.10196078, ..., 0.16470588,
         0.16470588, 0.13333333],
        [0.10196078, 0.10196078, 0.10196078, ..., 0.13333333,
         0.13333333, 0.13333333]],
```

```
        ...,

        [[0.02745098, 0.06666667, 0.1254902 , ..., 0.21176471,
          0.23921569, 0.37647059],
         [0.04705882, 0.05882353, 0.1254902 , ..., 0.24313725,
          0.22352941, 0.2745098 ],
         [0.07843137, 0.10588235, 0.09803922, ..., 0.22352941,
          0.23137255, 0.25882353],
         ...,
         [0.24313725, 0.25098039, 0.25098039, ..., 0.02745098,
          0.04705882, 0.04705882],
         [0.25098039, 0.26666667, 0.25098039, ..., 0.03529412,
          0.04705882, 0.04705882],
         [0.23137255, 0.25882353, 0.25098039, ..., 0.03529412,
          0.04705882, 0.04705882]],

        [[0.02745098, 0.06666667, 0.1254902 , ..., 0.21176471,
          0.23921569, 0.37647059],
         [0.04705882, 0.05882353, 0.1254902 , ..., 0.24313725,
          0.22352941, 0.2745098 ],
         [0.07843137, 0.10588235, 0.09803922, ..., 0.22352941,
          0.23137255, 0.25882353],
         ...,
         [0.24313725, 0.25098039, 0.25098039, ..., 0.02745098,
          0.04705882, 0.04705882],
         [0.25098039, 0.26666667, 0.25098039, ..., 0.03529412,
          0.04705882, 0.04705882],
         [0.23137255, 0.25882353, 0.25098039, ..., 0.03529412,
          0.04705882, 0.04705882]],

        [[0.02745098, 0.06666667, 0.1254902 , ..., 0.21176471,
          0.23921569, 0.37647059],
         [0.04705882, 0.05882353, 0.1254902 , ..., 0.24313725,
          0.22352941, 0.2745098 ],
         [0.07843137, 0.10588235, 0.09803922, ..., 0.22352941,
          0.23137255, 0.25882353],
         ...,
         [0.24313725, 0.25098039, 0.25098039, ..., 0.02745098,
          0.04705882, 0.04705882],
         [0.25098039, 0.26666667, 0.25098039, ..., 0.03529412,
          0.04705882, 0.04705882],
         [0.23137255, 0.25882353, 0.25098039, ..., 0.03529412,
          0.04705882, 0.04705882]]]])
```

```python
from tensorflow.keras.applications import ResNet50
from tensorflow.keras import layers, models
from tensorflow.keras.optimizers import Adam

images.shape
```

```
(3350, 224, 224)
```

```python
# Convert Labels to Binary Format
# Convert labels to binary integers (0 and 1)
labels = np.array(df['label'].values, dtype=np.int32)  # Assuming
labels are in the 'labels' column
print(np.unique(labels))  # Check if labels are [0, 1]
```

```
[0 1]
```

```python
image_shape = images.shape  # Shape of the images array

if len(image_shape) == 3 and image_shape[-1] == 1:
    print("The images are grayscale (1 channel).")
elif len(image_shape) == 3 and image_shape[-1] == 3:
    print("The images are RGB (3 channels).")
else:
    print("The image format is unknown or different.")
```

```
The image format is unknown or different.
```

```python
len(images.shape)
```

```
3
```

```python
images.shape[-1]
```

```
224
```

```python
# We need to convert them to RGB (3 channels) because the model
(ResNet50) expects 3-channel images as input.
if len(images.shape) == 3:  # images have shape (3350, 224, 224)
    # Add a channel dimension (grayscale to (3350, 224, 224, 1))
    images = np.expand_dims(images, axis=-1)

    # Convert grayscale images to RGB by repeating the single channel
3 times
    images_rgb = np.repeat(images, 3, axis=-1)  # Convert from (3350,
224, 224, 1) to (3350, 224, 224, 3)
else:
    images_rgb = images

print(images_rgb.shape)  # Should print (3350, 224, 224, 3)
```

```
(3350, 224, 224, 3)
```

```python
# Split data into training and validation sets (80% train, 20%
validation)
train_images, val_images, train_labels, val_labels = train_test_split(
    images_rgb, labels, test_size=0.2, random_state=42
)
```

```
print(train_images.shape, val_images.shape)  # Ensure it's split
correctly
```

```
(2680, 224, 224, 3) (670, 224, 224, 3)
```

```
# Training Data Generator (with Augmentation)
train_datagen = ImageDataGenerator(
    rescale=1./255,  # Normalize image pixels to [0, 1]
    rotation_range=30,  # Random rotations
    width_shift_range=0.2,  # Random horizontal shift
    height_shift_range=0.2,  # Random vertical shift
    shear_range=0.2,  # Random shear
    zoom_range=0.2,  # Random zoom
    horizontal_flip=True,  # Random horizontal flip
    fill_mode='nearest'  # Fill mode for new pixels
)

val_datagen = ImageDataGenerator(rescale=1./255)  # Only rescale for
validation

from tensorflow.keras.utils import Sequence
import numpy as np

# a custom data generator class CustomDataGenerator,
# which is a subclass of Keras's Sequence class.
# A data generator is typically used for feeding data in batches to a
machine learning model during training,
# especially when the dataset is too large to fit into memory

class CustomDataGenerator(Sequence):
    def __init__(self, images, labels, batch_size=32, shuffle=True):
        self.images = images
        self.labels = labels
        self.batch_size = batch_size
        self.shuffle = shuffle
        self.indexes = np.arange(len(self.images))
        self.index = 0  # Initialize the index here
        if self.shuffle:
            np.random.shuffle(self.indexes)

    def __len__(self):
        # Returns the number of batches per epoch.
        return int(np.floor(len(self.images) / self.batch_size))

    def on_epoch_end(self):
        # Shuffle the indexes after each epoch if shuffle is True
        # if there are 100 images and the batch size is 32, this will
return 100 // 32 = 3, meaning there are 3 full batches.
        if self.shuffle:
            np.random.shuffle(self.indexes)
```

```
    def __getitem__(self, index):
        # Generate one batch of data
        Purpose: This method retrieves one batch of data (both images
and labels).
        # How it works:
        # batch_indexes: This selects the appropriate indices for the
current batch.
        # For example, if batch_size=32 and index=2, it will select
the 64th to 95th image (32 * 2 to 32 * 3).
        # batch_images: Uses the batch_indexes to extract the images
for this batch.
        # batch_labels: Uses the batch_indexes to extract the labels
for this batch.
        # The method then returns the batch of images and labels.
        batch_indexes = self.indexes[index * self.batch_size:(index +
1) * self.batch_size]
        batch_images = self.images[batch_indexes]
        batch_labels = self.labels[batch_indexes]

        return batch_images, batch_labels

    def __iter__(self):
        # Makes the generator an iterator and initializes the index
        self.index = 0  # Reset the index at the start of each
iteration
        return self

    def __next__(self):
        # Returns the next batch of data
        if self.index < len(self):
            result = self.__getitem__(self.index)
            self.index += 1
            return result
        else:
            raise StopIteration

# Summary of What the Generator Does
# The generator takes in the dataset (images and labels), and
processes the data in batches.
# When iterating through the dataset, it returns batches of images and
labels by indexing into the dataset.
# After each epoch, the data is shuffled (if the shuffle flag is set
to True) to prevent the model from learning based on the order of the
data.
# The generator supports iteration, meaning it can be used in a for
loop or called using next().

# Create a custom generator for training and validation
train_generator = CustomDataGenerator(train_images, train_labels,
```

```
batch_size=16)
val_generator = CustomDataGenerator(val_images, val_labels,
batch_size=16)

# Check the output of the generator
x_batch, y_batch = next(train_generator)
print(x_batch.shape)  # Should print (batch_size, 224, 224, 3)
print(y_batch.shape)  # Should print (batch_size, )
```

```
(16, 224, 224, 3)
(16,)
```

```
# 1. Load the pre-trained ResNet50 model (excluding the top layers)
base_model = ResNet50(weights='imagenet', include_top=False,
input_shape=(224, 224, 3))

# 2. Freeze the base model (not trainable)
base_model.trainable = False

# 3. Add custom layers on top of ResNet50
# Model should have output layer for binary classification
model = models.Sequential([
    base_model,  # Pre-trained ResNet50 layers
    layers.GlobalAveragePooling2D(),  # Global average pooling to
reduce output size
    layers.Dense(1024, activation='relu'),  # Fully connected layer
    layers.Dropout(0.5),  # Dropout for regularization
    layers.Dense(1, activation='sigmoid')  # Output layer for binary
classification
])

# 4. Compile the model
model.compile(optimizer=Adam(), loss='binary_crossentropy',
metrics=['accuracy'])

# Train the model
history = model.fit(
    train_generator,  # The training data generator
    epochs=10,  # Number of epochs to train
    validation_data=val_generator  # The validation data generator
)
```

```
C:\Users\DELL\anaconda3\Lib\site-packages\keras\src\trainers\
data_adapters\py_dataset_adapter.py:121: UserWarning: Your `PyDataset`
class should call `super().__init__(**kwargs)` in its constructor.
`**kwargs` can include `workers`, `use_multiprocessing`,
`max_queue_size`. Do not pass these arguments to `fit()`, as they will
be ignored.
  self._warn_if_super_not_called()
```

```
Epoch 1/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 231s 1s/step - accuracy: 0.9613 - loss:
0.2078 - val_accuracy: 0.9680 - val_loss: 0.1723
Epoch 2/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 223s 1s/step - accuracy: 0.9770 - loss:
0.1265 - val_accuracy: 0.9680 - val_loss: 0.1993
Epoch 3/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 219s 1s/step - accuracy: 0.9812 - loss:
0.0979 - val_accuracy: 0.9680 - val_loss: 0.1656
Epoch 4/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 237s 1s/step - accuracy: 0.9779 - loss:
0.1121 - val_accuracy: 0.9680 - val_loss: 0.1371
Epoch 5/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 227s 1s/step - accuracy: 0.9842 - loss:
0.0855 - val_accuracy: 0.9680 - val_loss: 0.1306
Epoch 6/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 236s 1s/step - accuracy: 0.9868 - loss:
0.0742 - val_accuracy: 0.9710 - val_loss: 0.1232
Epoch 7/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 234s 1s/step - accuracy: 0.9830 - loss:
0.0893 - val_accuracy: 0.9695 - val_loss: 0.1237
Epoch 8/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 221s 1s/step - accuracy: 0.9770 - loss:
0.1046 - val_accuracy: 0.9680 - val_loss: 0.1388
Epoch 9/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 221s 1s/step - accuracy: 0.9828 - loss:
0.0861 - val_accuracy: 0.9680 - val_loss: 0.1375
Epoch 10/10
167/167 ━━━━━━━━━━━━━━━━━━━━ 217s 1s/step - accuracy: 0.9823 - loss:
0.0880 - val_accuracy: 0.9680 - val_loss: 0.1629
```

Training Accuracy: Your model's training accuracy is high, ranging from 97.7% to 98.6%, which suggests it's learning effectively from the training data.

Training Loss: The training loss is relatively low (ranging from 0.0727 to 0.1323), indicating the model is minimizing the loss function well.

Validation Accuracy: The validation accuracy is consistently around 96.8%, which is a good sign of the model generalizing well to unseen data. However, it has not improved much beyond this value across epochs, which may indicate that the model is reaching its peak performance or that further training does not lead to significant improvement.

Validation Loss: The validation loss fluctuates between 0.1306 to 0.2365 and is higher than the training loss, which is common. It indicates some overfitting but not drastically, given the validation accuracy is still high.

Evaluate Model Performance

```
# We can first evaluate the model using the val_generator
# This will give us a quick look at the model's performance on unseen data
# 1. Evaluating the Model on the Validation Set:
val_loss, val_accuracy = model.evaluate(val_generator)
print(f'Validation Loss: {val_loss:.4f}')
print(f'Validation Accuracy: {val_accuracy:.4f}')
```

```
41/41 ━━━━━━━━━━━━━━━━━━━━ 40s 968ms/step - accuracy: 0.9612 - loss: 0.1994
Validation Loss: 0.1629
Validation Accuracy: 0.9680
```

```
# 2. Predicting on the Validation Set:
# We can also predict the labels for the validation set and compute metrics such as precision, recall, F1-score, and confusion matrix.
# Predict on the validation set
val_predictions = model.predict(val_images)
val_predictions = (val_predictions > 0.5)  # Convert probabilities to binary predictions
```

```
21/21 ━━━━━━━━━━━━━━━━━━━━ 45s 2s/step
```

```
val_predictions[:5]
```

```
array([[ True],
       [ True],
       [ True],
       [ True],
       [ True]])
```

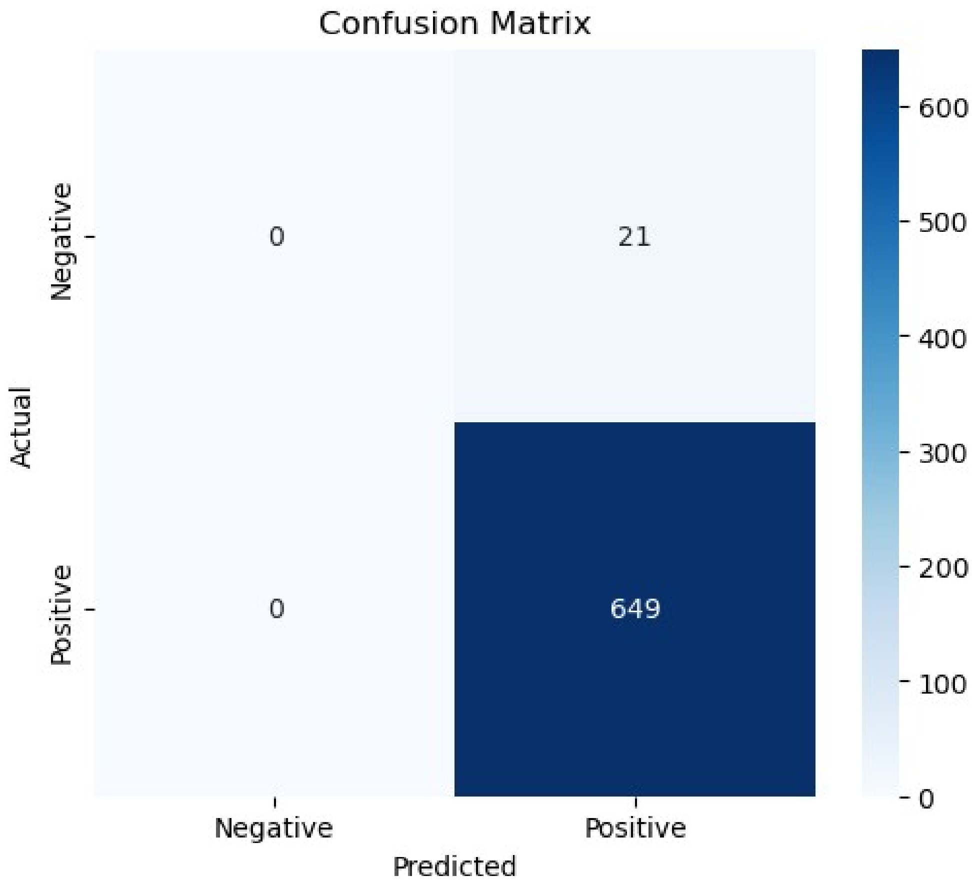
```
# Calculate precision, recall, F1-score
from sklearn.metrics import precision_score, recall_score, f1_score, confusion_matrix

precision = precision_score(val_labels, val_predictions)
recall = recall_score(val_labels, val_predictions)
f1 = f1_score(val_labels, val_predictions)
print(f'Precision: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1-Score: {f1:.4f}')
```

```
Precision: 0.9687
Recall: 1.0000
F1-Score: 0.9841
```

```
# Confusion matrix
cm = confusion_matrix(val_labels, val_predictions)
print(f'Confusion Matrix:\n{cm}')
```

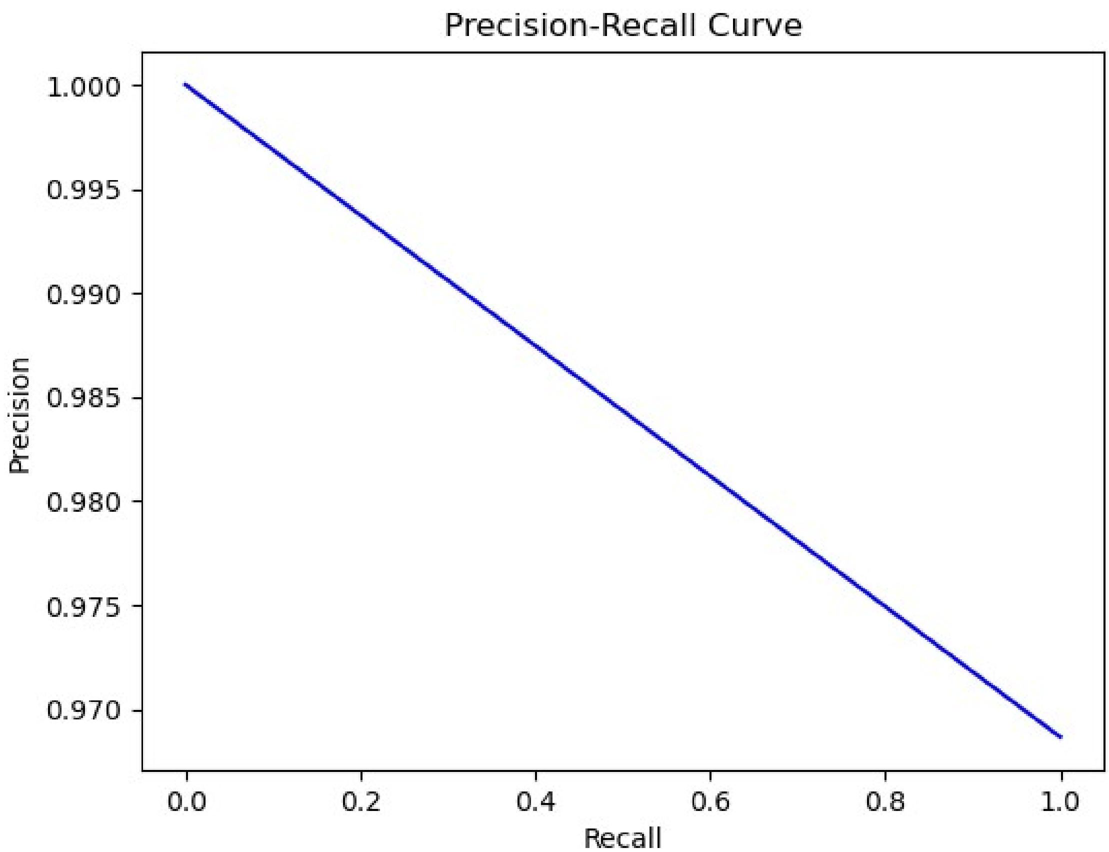
```
Confusion Matrix:
[[  0  21]
 [  0 649]]

[[  0  21]   # True class 0 (negative) | Predicted as class 0,
Predicted as class 1
 [  0 649]]  # True class 1 (positive) | Predicted as class 0,
Predicted as class 1
# Explanation:
# True Negatives (TN) = 0:
# There were no instances of class 0 (negative class) that were
correctly predicted as class 0.

# False Positives (FP) = 21:
# There are 21 instances of class 0 (negative class) that the model
incorrectly predicted as class 1 (positive class). These are false
positives.

# False Negatives (FN) = 0:
# There are no instances of class 1 (positive class) that were
incorrectly predicted as class 0. The model has perfectly captured all
the positive cases.

# True Positives (TP) = 649:
# The model correctly predicted 649 instances of class 1 (positive
class) as class 1. These are true positives.

# Summary:
# The model missed 21 negative samples (false positives) by predicting
them as positive (class 1).
# However, all positive samples (class 1) were correctly predicted as
positive.
# No false negatives were made, which means the model did not miss any
actual positives.
# The model has performed well overall, but there’s a slight issue
with false positives, where it incorrectly predicted negatives as
positives.

  Cell In[175], line 1
    [[  0  21]   # True class 0 (negative) | Predicted as class 0,
Predicted as class 1
        ^
SyntaxError: invalid syntax. Perhaps you forgot a comma?


# 3. Visualize the Performance:
import matplotlib.pyplot as plt
import seaborn as sns

# Plot Confusion Matrix
plt.figure(figsize=(6, 5))
```

```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=['Negative', 'Positive'], yticklabels=['Negative',
'Positive'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()
```



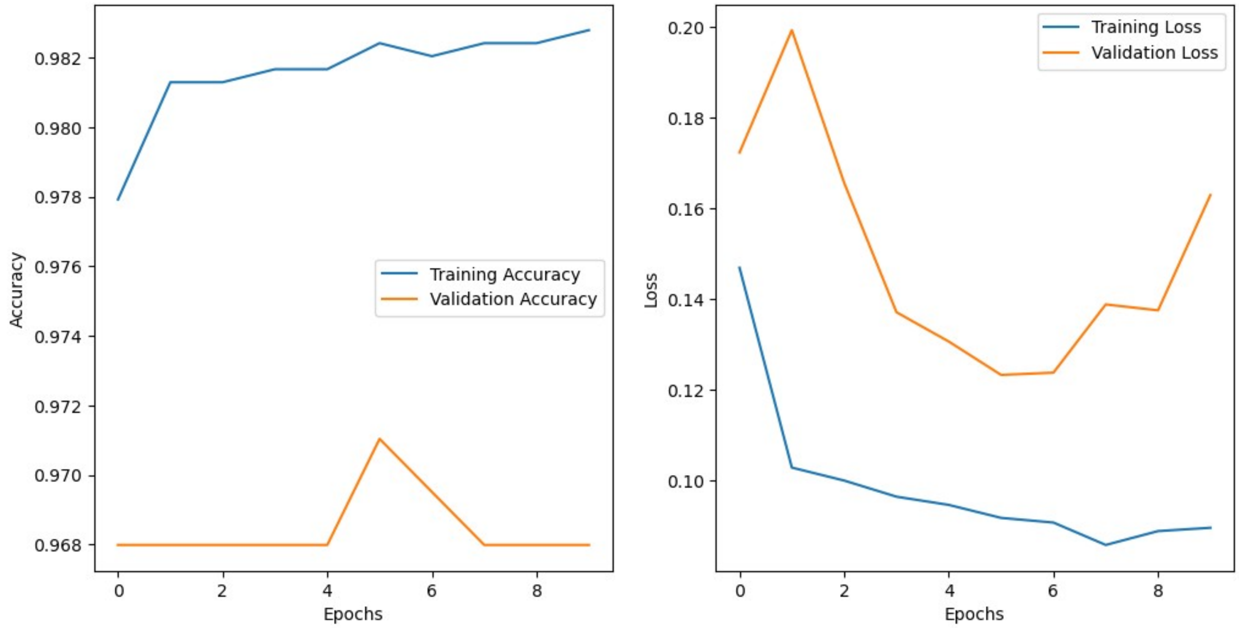

```
# . Plot Precision-Recall Curve:
# If you want to get a better idea of the model's performance across
different thresholds, you can plot the precision-recall curve.
from sklearn.metrics import precision_recall_curve

# Compute precision-recall curve
precision, recall, thresholds = precision_recall_curve(val_labels,
val_predictions)

# Plot precision-recall curve
plt.plot(recall, precision, color='b')
plt.xlabel('Recall')
plt.ylabel('Precision')
```

```
plt.title('Precision-Recall Curve')
plt.show()
```




```
# 5. Check for Overfitting:
# Assuming you saved the `history` object during training
plt.figure(figsize=(12, 6))

# Plot training and validation accuracy
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Plot training and validation loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

```
plt.show()
```



Saving the Model in Jupyter Notebook:

```
model.save('face_detection_model_final_1.keras')
```