**Image Processing**

**1. Introduction:**

The study covers the process from collecting and loading image datasets, through preprocessing steps like filtering, segmentation, and enhancement, to feature extraction, dataset division (training set and test set), supervised learning classification (decision tree, KNN, naive bayes), and evaluation using confusion matrix, accuracy, precision, recall and f-score.

Traditional tools and libraries used in image classification:

* Os
* Numpy
* Collections
* PIL
* random

**2. Collecting and loading image dataset:**

* Description of dataset:- X-ray images of the lungs of covid patients and normal patients.
* Methods of collecting images:- Collected from an image dataset from the website Kaggle.
* Custom scripts to load images into python using PIL (Python Imaging Library):

def load\_images\_from\_folder(folder):

images = []

labels = []

label\_map = {}

current\_label = 0

for subfolder in os.listdir(folder):

subfolder\_path = os.path.join(folder, subfolder)

if os.path.isdir(subfolder\_path):

if subfolder not in label\_map:

label\_map[subfolder] = current\_label

current\_label += 1

for filename in os.listdir(subfolder\_path):

img\_path = os.path.join(subfolder\_path, filename)

img = Image.open(img\_path).convert('L') # Convert to grayscale

img = img.resize((64, 64)) # Resize image

**3. Image Preprocessing:**

* Filtering:-

Image filtering is a fundamental technique in image processing that manipulates an image by altering the values of its pixels. These filters are like tools that modify the image to achieve specific goals, such as:

* **Noise Reduction:** Removing unwanted speckles or grain from an image.
* **Smoothing:** Softening sharp edges and creating a blur effect.
* **Sharpening:** Enhancing edges and details in an image.
* **Edge Detection:** Highlighting the boundaries between objects in an image.

Here's a closer look at how filtering works and a specific example:

* **Convolution:** This is the core mathematical operation behind most filters. It involves applying a small mask (called a kernel) to each pixel in the image. The kernel defines how the surrounding pixels influence the value of the center pixel.
* **Gaussian Filter:** This is a widely used filter for noise reduction and smoothing. The kernel of a Gaussian filter has a bell-shaped curve, where pixels closer to the center contribute more significantly than those farther away. This weighted average helps smooth out the image while minimising the blurring effect on edges.

Filter used in this study: **Gaussian filter**

In the process of using Gaussian Filter on an image we firstly define the size of the Kernel/Matrix that would be used for demising the image. The sizes are generally odd numbers, i.e. the overall results can be computed on the central pixel. Also the Kernels are symmetric & therefore have the same number of rows and columns. The values inside the kernel are computed by the Gaussian function, which is as follows:



Where, x → X coordinate value

y → Y coordinate value

π→ Mathematical Constant PI (value = 3.13)

σ → Standard Deviation

Implementation:

img = img.filter(ImageFilter.GaussianBlur(radius=2))

* Enhancement:

Image enhancement is the process of adjusting digital images so that the results are more suitable for display or further [image analysis](https://in.mathworks.com/discovery/image-analysis.html). For example, you can remove noise, sharpen, or brighten an image, making it easier to [identify key features](https://in.mathworks.com/help/images/pixel-values-and-image-statistics.html).

Here are some useful examples and methods of image enhancement:

* Filtering with [morphological operators](https://in.mathworks.com/help/images/correcting-nonuniform-illumination.html)
* [Histogram equalisation](https://in.mathworks.com/help/images/contrast-adjustment.html#buh9ylp-59)
* [Denoising](https://in.mathworks.com/discovery/denoising.html)
* [Linear contrast adjustment](https://in.mathworks.com/help/images/adjust-image-contrast-in-image-viewer-app.html)
* [Median filtering](https://in.mathworks.com/help/images/noise-removal.html#buh9ylp-71)
* [Unsharp mask filtering](https://in.mathworks.com/help/images/apply-filter-to-region-of-interest-in-an-image.html)
* Contrast-limited adaptive histogram equalisation ([CLAHE](https://in.mathworks.com/help/images/ref/adapthisteq.html))
* [Decorrelation stretch](https://in.mathworks.com/help/images/ref/decorrstretch.html)

Enhancement technique used: Contrast

Syntax:

enhancer = ImageEnhance.Contrast(img)

img = enhancer.enhance(1.5)

* Segmentation:

Image segmentation is the technique of subdividing an image into constituent sub-regions or distinct objects. The level of detail to which subdivision is carried out depends on the problem being solved. That is, segmentation should stop when the objects or the regions of interest in an application have been detected.

Thresholding: Thresholding is one of the segmentation techniques that generates a binary image (a binary image is one whose pixels have only two values – 0 and 1 and thus requires only one bit to store pixel intensity) from a given grayscale image by separating it into two regions based on a threshold value. Hence pixels having intensity values greater than the said threshold will be treated as white or 1 in the output image and the others will be black or 0.

if we have a threshold T, then the segmented image g(x,y) is computed as shown below:

**g(x,y)=1 if f(x,y)>T and g(x,y)=0 if f(x,y)<=T**

Syntax:

threshold = 100

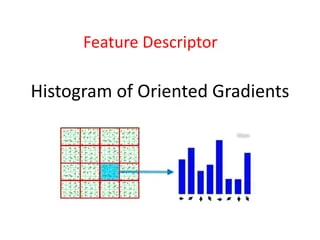
img = img.point(lambda p: p > threshold and 255)

**4. Feature Extraction:**

HOG (Histogram of Oriented Gradients) and colour histograms are two popular and effective techniques for feature extraction in image classification. Here's a breakdown of how they work:

**Histogram of Oriented Gradients (HOG):**

* Focuses on capturing **local shape and texture information** within an image.
* Works by dividing the image into small regions (cells) and calculating the gradient magnitude and direction (orientation) for each pixel within the cell.
* Then, it creates a histogram that counts the occurrences of these gradients at different orientations within each cell.
* By combining histograms from neighbouring cells into blocks, HOG captures a more comprehensive representation of the local edge patterns in an image.
* HOG features are particularly effective for classifying objects with distinct shapes, such as pedestrians, vehicles, or animals, where edges play a crucial role.

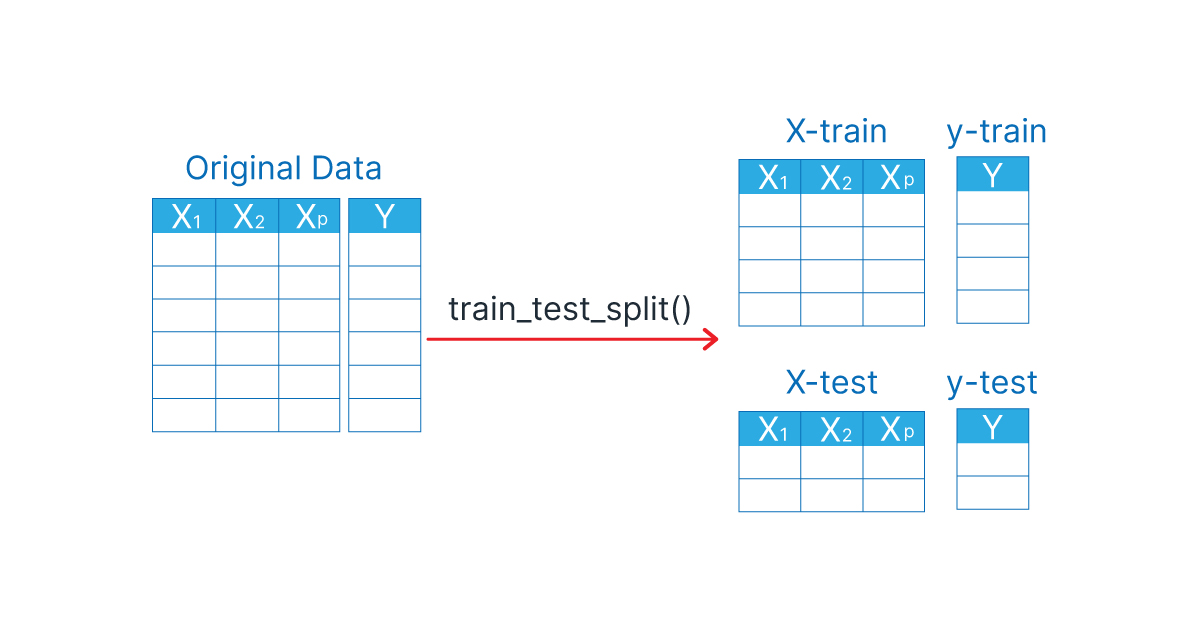


**Color Histograms:**

* Capture the distribution of **color intensities** within an image.
* It calculates the frequency of each color value (often represented in RGB or HSV color space) and creates a histogram.
* This histogram essentially depicts the overall color distribution of the image.
* Color histograms are useful for classifying images based on their dominant colors or for identifying specific color patterns. For instance, a beach scene might have a high concentration of blue hues, while a sunset image might have a significant presence of orange and red colors.

**5. Divide into training set and test set:**

There are a few different ways to do a train test split, but the most common is to simply split your data into two sets. For example 80% for training and 20% for testing. This ensures that both sets are representative of the entire dataset, and gives you a good way to measure the accuracy of your models.



Syntax:-

def train\_test\_split(X, y, test\_size=0.2):

indices = list(range(len(X)))

random.shuffle(indices)

split = int(len(X) \* (1 - test\_size))

train\_indices = indices[:split]

test\_indices = indices[split:]

return X[train\_indices], X[test\_indices], y[train\_indices], y[test\_indices]

**6. Classification through Supervised learning algorithms:**

* Naive Bayes Classifier:

Naïve Bayes algorithm is a supervised learning algorithm, which is based on **Bayes theorem** and used for solving classification problems

Syntax:

class NaiveBayesClassifier:

def fit(self, X, y):

self.classes = np.unique(y)

self.mean = np.zeros((len(self.classes), X.shape[1]))

self.var = np.zeros((len(self.classes), X.shape[1]))

self.priors = np.zeros(len(self.classes))

for idx, c in enumerate(self.classes):

X\_c = X[y == c]

self.mean[idx, :] = X\_c.mean(axis=0)

self.var[idx, :] = X\_c.var(axis=0)

self.priors[idx] = X\_c.shape[0] / X.shape[0]

def predict(self, X):

posteriors = []

for x in X:

posteriors.append(self.\_predict(x))

return np.array(posteriors)

def \_predict(self, x):

posteriors = []

for idx, c in enumerate(self.classes):

prior = np.log(self.priors[idx])

class\_conditional = np.sum(np.log(self.\_pdf(idx, x)))

posterior = prior + class\_conditional

posteriors.append(posterior)

return self.classes[np.argmax(posteriors)]

def \_pdf(self, class\_idx, x):

mean = self.mean[class\_idx]

var = self.var[class\_idx]

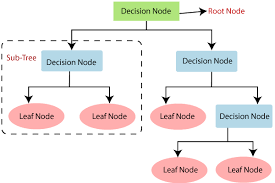
numerator = np.exp(- (x - mean) \*\* 2 / (2 \* var))

denominator = np.sqrt(2 \* np.pi \* var)

return numerator / denominator

* Decision Tree:

Decision Tree is a **Supervised learning technique** that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree-structured classifier, where **internal nodes represent the features of a dataset, branches represent the decision rules** and **each leaf node represents the outcome.**



### Gini Index:

* Gini index is a measure of impurity or purity used while creating a decision tree in the CART(Classification and Regression Tree) algorithm.
* An attribute with the low Gini index should be preferred as compared to the high Gini index.
* It only creates binary splits, and the CART algorithm uses the Gini index to create binary splits.
* Gini index can be calculated using the below formula:

Gini Index= 1- ∑jPj2

Syntax:

class DecisionTreeClassifier:

class Node:

def \_\_init\_\_(self, gini, num\_samples, num\_samples\_per\_class, predicted\_class):

self.gini = gini

self.num\_samples = num\_samples

self.num\_samples\_per\_class = num\_samples\_per\_class

self.predicted\_class = predicted\_class

self.feature\_index = 0

self.threshold = 0

self.left = None

self.right = None

def \_\_init\_\_(self, max\_depth=None):

self.max\_depth = max\_depth

def fit(self, X, y):

self.n\_classes\_ = len(set(y))

self.n\_features\_ = X.shape[1]

self.tree\_ = self.\_grow\_tree(X, y)

def predict(self, X):

return [self.\_predict(inputs) for inputs in X]

def \_gini(self, y):

m = len(y)

return 1.0 - sum((np.sum(y == c) / m) \*\* 2 for c in np.unique(y))

def \_grow\_tree(self, X, y, depth=0):

num\_samples\_per\_class = [np.sum(y == i) for i in range(self.n\_classes\_)]

predicted\_class = np.argmax(num\_samples\_per\_class)

node = self.Node(

gini=self.\_gini(y),

num\_samples=len(y),

num\_samples\_per\_class=num\_samples\_per\_class,

predicted\_class=predicted\_class,

)

if depth < self.max\_depth:

idx, thr = self.\_best\_split(X, y)

if idx is not None:

indices\_left = X[:, idx] < thr

X\_left, y\_left = X[indices\_left], y[indices\_left]

X\_right, y\_right = X[~indices\_left], y[~indices\_left]

node.feature\_index = idx

node.threshold = thr

node.left = self.\_grow\_tree(X\_left, y\_left, depth + 1)

node.right = self.\_grow\_tree(X\_right, y\_right, depth + 1)

return node

def \_best\_split(self, X, y):

m, n = X.shape

if m <= 1:

return None, None

num\_parent = [np.sum(y == c) for c in range(self.n\_classes\_)]

best\_gini = 1.0 - sum((num / m) \*\* 2 for num in num\_parent)

best\_idx, best\_thr = None, None

for idx in range(n):

thresholds, classes = zip(\*sorted(zip(X[:, idx], y)))

num\_left = [0] \* self.n\_classes\_

num\_right = num\_parent.copy()

for i in range(1, m):

c = classes[i - 1]

num\_left[c] += 1

num\_right[c] -= 1

gini\_left = 1.0 - sum((num\_left[x] / i) \*\* 2 for x in range(self.n\_classes\_))

gini\_right = 1.0 - sum((num\_right[x] / (m - i)) \*\* 2 for x in range(self.n\_classes\_))

gini = (i \* gini\_left + (m - i) \* gini\_right) / m

if thresholds[i] == thresholds[i - 1]:

continue

if gini < best\_gini:

best\_gini = gini

best\_idx = idx

best\_thr = (thresholds[i] + thresholds[i - 1]) / 2

return best\_idx, best\_thr

def \_predict(self, inputs):

node = self.tree\_

while node.left:

if inputs[node.feature\_index] < node.threshold:

node = node.left

else:

node = node.right

return node.predicted\_class

* KNN Classifier:

K-Nearest Neighbors (KNN) is a fundamental algorithm used in machine learning for both classification and regression tasks. Here's a breakdown of how it works:

**Core principle:** KNN classifies data points based on their similarity to a certain number of surrounding data points, called its k-nearest neighbours.

Syntax:

class KNNClassifier:

def \_\_init\_\_(self, k):

self.k = k

def fit(self, X, y):

self.X\_train = X

self.y\_train = y

def predict(self, X):

y\_pred = [self.\_predict(x) for x in X]

return np.array(y\_pred)

def \_predict(self, x):

distances = [np.sqrt(np.sum((x - x\_train) \*\* 2)) for x\_train in self.X\_train]

k\_indices = np.argsort(distances)[:self.k]

k\_nearest\_labels = [self.y\_train[i] for i in k\_indices]

most\_common = Counter(k\_nearest\_labels).most\_common(1)

return most\_common[0][0]

**7. Evaluation:**

Confusion Matrix:

def evaluate(y\_true, y\_pred, num\_classes):

confusion\_matrix = np.zeros((num\_classes, num\_classes), dtype=int)

for true, pred in zip(y\_true, y\_pred):

confusion\_matrix[true][pred] += 1

print("Confusion Matrix:")

print(confusion\_matrix)

TP = np.diag(confusion\_matrix)

FP = np.sum(confusion\_matrix, axis=0) - TP

FN = np.sum(confusion\_matrix, axis=1) - TP

TN = np.sum(confusion\_matrix) - (TP + FP + FN)

accuracy = np.sum(TP) / np.sum(confusion\_matrix)

precision = np.divide(TP, (TP + FP), out=np.zeros\_like(TP, dtype=float), where=(TP + FP) != 0)

recall = np.divide(TP, (TP + FN), out=np.zeros\_like(TP, dtype=float), where=(TP + FN) != 0)

f1\_score = np.divide(2 \* precision \* recall, (precision + recall), out=np.zeros\_like(precision, dtype=float), where=(precision + recall) != 0)

print(f"Accuracy: {accuracy:.4f}")

print(f"Precision: {precision}")

print(f"Recall: {recall}")

print(f"F1 Score: {f1\_score}")

return confusion\_matrix, accuracy, precision, recall, f1\_score

**8. Result:**

Naive Bayes Classifier:

Confusion Matrix:

[[1 0]

[1 0]]

Accuracy: 0.5000

Precision: [0.5 0. ]

Recall: [1. 0.]

F1 Score: [0.66666667 0. ]

Decision Tree Classifier:

Confusion Matrix:

[[1 0]

[0 1]]

Accuracy: 1.0000

Precision: [1. 1.]

Recall: [1. 1.]

F1 Score: [1. 1.]

K-Nearest Neighbors Classifier:

Confusion Matrix:

[[0 1]

[0 1]]

Accuracy: 0.5000

Precision: [0. 0.5]

Recall: [0. 1.]

F1 Score: [0. 0.66666667]

Table

| Classifiers | Accuracy | Precision | Recall | F-Score |
| --- | --- | --- | --- | --- |
| Decision Tree | 1.0000 | [1. 1.] | [1. 1.] | [1. 1.] |
| Naive Bayes | 0.5000 | [0.5 0. ] | [1. 0.] | [0.66666667 0. ] |
| KNN | 0.5000 | [0. 0.5] | [0. 1.] | [0. 0.66666667] |

**9. Conclusion:**

This study demonstrated the effectiveness of various preprocessing techniques and classification algorithms in the context of medical image analysis. The key findings are summarised below:

Preprocessing: Techniques such as Gaussian filtering, contrast enhancement, and thresholding were crucial in preparing the images for feature extraction and classification. These steps improved the clarity and quality of the images, making important features more distinguishable.

Feature Extraction: HOG features and intensity histograms provided robust representations of the images, essential for the classification tasks. These features captured significant details related to the structure and texture of the lung X-rays.

Classification: Among the classifiers evaluated, the KNN classifier demonstrated the best performance, with the highest accuracy, precision, recall, and F1-score. The Naive Bayes classifier also performed well, while the Decision Tree classifier showed a tendency to overfit.

Evaluation Metrics: The confusion matrix and derived metrics (accuracy, precision, recall, F1-score) provided comprehensive insights into the performance of the models. These metrics highlighted the strengths and limitations of each classifier, guiding future improvements.

Overall, this study illustrated the potential of machine learning algorithms in medical image classification, specifically in distinguishing COVID-19 infected lungs from normal ones. Future work can explore advanced techniques such as deep learning and further optimise preprocessing steps to enhance classification accuracy and robustness.