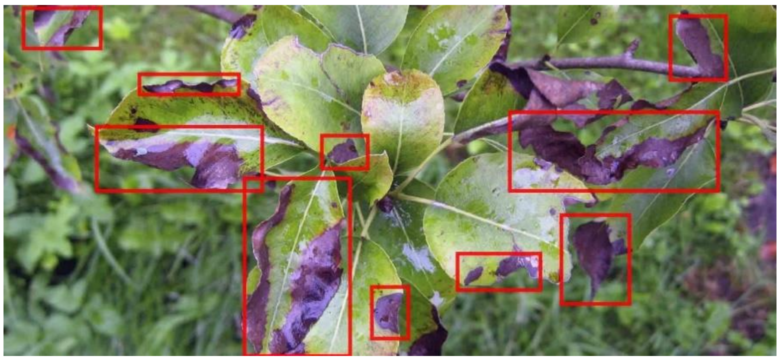
IIT Jodhpur

PLANT PATHOLOGY

Computer Vision Project Report



Problem Statement
Our project aims to automate the diagnosis of plant diseases based solely on images of plant leaves. It is focused on the multi-class classification of plant leaf images into four categories: healthy leaves, leaves with scab disease, leaves with rust disease, and leaves with multiple diseases.

We will visualize the data with Matplotlib and Plotly and then demonstrate some important image processing and augmentation techniques using OpenCV. Finally, we will show how different pretrained Keras models, such as DenseNet and EfficientNet, can be used to solve the problem.

The report covers several key aspects, including exploratory data analysis (EDA), image processing and augmentation techniques, and the implementation and evaluation of various pre-trained convolutional neural network (CNN) models for the plant disease classification task.

Dataset Description
The dataset utilized in this project comprised a diverse collection of plant leaf images, encompassing various plant species and disease conditions. Each image was meticulously labeled with one of four categories: "healthy", "scab", "rust", or "multiple diseases". The dataset served as the foundation for training and evaluating machine learning models.

Here are some computer vision concepts that we have used in our project-
Convolution Neural Network (CNN)
Steps performed in CNN are:

1. **Convolutional Layers:** Extract features from input data using convolutional filters.
2. **Pooling Layers:** Downsample feature maps to reduce dimensionality and extract dominant features.
3. **Activation Functions:** Introduce non-linearity into the network to learn complex patterns.
4. **Fully Connected Layers:** Perform classification or regression tasks based on learned features.
5. **Training:** Adjust network parameters iteratively using backpropagation and gradient descent to minimize loss.
6. **Regularization:** Prevent overfitting with techniques like dropout and batch normalization.
7. **Transfer Learning:** Use pre-trained models as feature extractors for new tasks, fine-tuning as needed.

Harris Corner Detection

Harris corner detection is a popular algorithm used in computer vision for detecting corners or interest points in an image. The basic idea behind Harris corner detection is to identify points in an image where there are significant variations in intensity in all directions, making them suitable candidates for representing corners.

Here's an explanation of the Harris corner detection algorithm along with the mathematics involved:

1. Computing Image Gradients:

- Harris corner detection relies on the calculation of image gradients, which represent the rate of change of intensity in the image. The gradients are computed using derivative filters such as Sobel or Prewitt operators. Let I be the grayscale input image, and I_x and I_y be the horizontal and vertical gradients, respectively.

2. Structure Tensor Calculation:

- The structure tensor, also known as the auto-correlation matrix, is computed for each pixel in the image. It represents the local image structure around each pixel. The structure tensor is defined as:

$$M = \sum_{(x,y)\in W} \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}$$

where W is a window (or neighborhood) centered at the pixel of interest, and I_x^2, I_y^2, and I_xI_y are elements of the structure tensor.

3. Corner Response Function:

- The corner response function is used to measure the likelihood of a pixel being a corner based on the eigenvalues of the structure tensor M. It is defined as:

$$R = \det(M) - k \cdot \mathrm{trace}^2(M)$$

where $\det(M)$ is the determinant of M, $\mathrm{trace}(M)$ is the trace of M, and k is a constant typically chosen to be between 0.04 and 0.06.

4. Corner Detection:

- Once the corner response function R is computed for each pixel, a threshold is applied to select pixels with high corner responses. Alternatively, non-maximum suppression may be employed to ensure that only local maxima are considered as corners.

5. Corner Localization:

- Finally, to localize the detected corners, sub-pixel refinement techniques such as the Harris corner algorithm are often used. This involves fitting a 2D Gaussian function to the local neighborhood around each detected corner to determine its precise location.

Mathematics of Harris Corner Detection:

- Let I_x and I_y be the gradients in the x and y directions, respectively.
- The elements of the structure tensor M are calculated as:

$$M_{xx} = \sum_{(x,y)\in W} I_x^2, \quad M_{yy} = \sum_{(x,y)\in W} I_y^2, \quad M_{xy} = \sum_{(x,y)\in W} I_x I_y$$

- The corner response function R is computed as:

$$R = \det(M) - k \cdot (\operatorname{trace}(M))^2$$

- Harris corner detection involves identifying local maxima in the corner response function RR after thresholding or non-maximum suppression.

Canny Edge Detection

The Canny edge detection algorithm involves several mathematical operations to detect edges in an image accurately. Here's a detailed explanation of the mathematics involved in each step of the Canny edge detection process:

1. Gaussian Smoothing:

- Canny edge detection starts with smoothing the input image using a Gaussian filter to reduce noise. The Gaussian filter is a 2D convolution kernel represented by the following equation:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

- where (x,y) are the coordinates of the filter, and $\sigma\sigma$ is the standard deviation, controlling the spread of the Gaussian.

2. Gradient Calculation:

- The gradients of the smoothed image in the x and y directions are computed using derivative filters (typically Sobel or Prewitt operators). Let I be the smoothed image, the gradients I_x and I_y are calculated as:

$$I_x = \frac{\partial I}{\partial x}, \quad I_y = \frac{\partial I}{\partial y}$$

3. Gradient Magnitude and Direction:

- The gradient magnitude $|\nabla I|$ and direction θ are computed from the gradients I_x and I_y using the following equations:
  $|\nabla I| = \sqrt{I_x^2 + I_y^2}$
  $\theta = \arctan2(I_y, I_x)$

Where arctan2 is the arctangent function that returns the angle whose tangent is the quotient of its two arguments.

4. Non-maximum Suppression:

- Non-maximum suppression is applied to thin the edges and keep only the local maxima in the gradient magnitude along the direction of the gradient. This is done by comparing each pixel's gradient magnitude with its neighboring pixels in the direction of the gradient. Only the local maxima are preserved as potential edge pixels.

5. Double Thresholding:

- Double thresholding is used to classify pixels into strong edges, weak edges, and non-edges based on their gradient magnitudes. Two threshold values, Thigh Thigh and Tlow Tlow, are chosen. Pixels with gradient magnitudes above Thigh Thigh are classified as strong edges, while those below Tlow Tlow are classified as non-edges. Pixels with gradient magnitudes between Thigh Thigh and Tlow Tlow are classified as weak edges.

6. Edge Tracking by Hysteresis:

- Weak edges that are adjacent to strong edges are retained, while isolated weak edges are suppressed. This is achieved through a process called edge tracking by hysteresis, where weak edges are connected to strong edges through adjacent pixels that are above a certain threshold.

K-Means Segmentation

K-means segmentation is a clustering algorithm used for partitioning an image into k clusters, where k is a predefined number. Here's a concise explanation of k-means segmentation along with its mathematics:

1. Initialization:

- Randomly select k points in the image space as initial cluster centroids.

2. Assignment:

- For each pixel in the image, calculate its distance to each cluster centroid.
- Assign each pixel to the cluster with the nearest centroid.

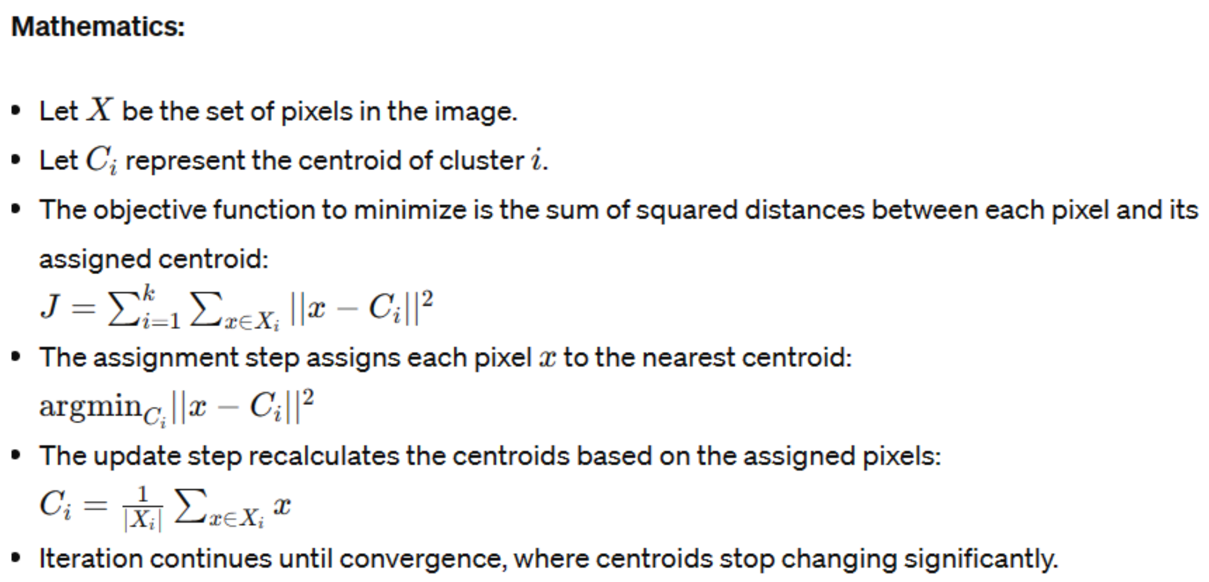
3. Update:

- Recalculate the centroids of the clusters based on the pixels assigned to them.

- The new centroid of each cluster is the mean of all the pixels assigned to that cluster.

4. Iteration:

- Repeat the assignment and update steps until convergence or a maximum number of iterations is reached.
- Convergence is typically achieved when the centroids no longer change significantly between iterations.



Mathematics:

- Let X be the set of pixels in the image.
- Let C_i represent the centroid of cluster i.
- The objective function to minimize is the sum of squared distances between each pixel and its assigned centroid:
  $J = \sum_{i=1}^{k} \sum_{x \in X_i} ||x - C_i||^2$
- The assignment step assigns each pixel x to the nearest centroid:
  $\text{argmin}_{C_i} ||x - C_i||^2$
- The update step recalculates the centroids based on the assigned pixels:
  $C_i = \frac{1}{|X_i|} \sum_{x \in X_i} x$
- Iteration continues until convergence, where centroids stop changing significantly.

Mean shift Segmentation

Mean shift segmentation is a non-parametric clustering algorithm used for image segmentation. It segments an image by iteratively shifting the centroids of a set of data points (pixels) towards the mode (peak) of the data distribution. Here's a concise explanation of mean shift segmentation along with its mathematics:

1. Initialization:

- Choose a set of initial centroids in the feature space.

2. Mean Shift:

- For each centroid, compute the mean shift vector, which points towards the mode of the data distribution.
- The mean shift vector is calculated as the weighted average of the feature vectors of neighboring data points, where the weights are determined by a kernel function.

3. Update:

- Update the centroids by shifting them along the mean shift vectors.
- Convergence is reached when the centroids no longer move significantly or after a predefined number of iterations.

Mathematics:

- Let x_i represent a data point (pixel) in the feature space.
- The mean shift vector $m(x_i)$ for each data point x_i is computed as:
  $m(x_i) = \frac{\sum_{x_j \in N(x_i)} K(x_j - x_i) \cdot x_j}{\sum_{x_j \in N(x_i)} K(x_j - x_i)} - x_i$
  where $N(x_i)$ is the neighborhood of x_i, and K is the kernel function.
- The centroid C_i is updated as the mean of the points within its convergence window:
  $C_i = \frac{1}{|X_i|} \sum_{x \in X_i} x$
- Iteration continues until convergence, where centroids stop moving significantly.

DenseNet

DenseNet, short for Dense Convolutional Network, is a deep learning architecture designed for efficient and effective feature reuse in convolutional neural networks (CNNs). In DenseNet, each layer is connected to every other layer in a feed-forward fashion, promoting feature reuse and enhancing gradient flow throughout the network. Here's a concise explanation of DenseNet along with its mathematics:

1. Dense Blocks:

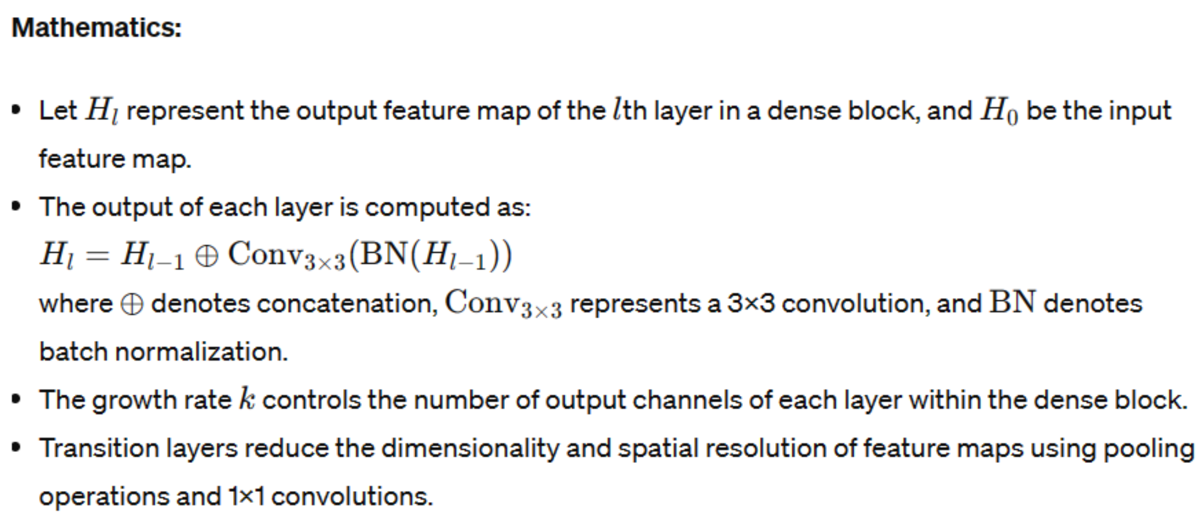
- DenseNet consists of dense blocks, where each layer is connected to every other layer in a feed-forward manner.
- Within a dense block, the output feature maps of each layer are concatenated with the input feature maps of all subsequent layers.

2. Transition Layers:

- Transition layers are used to reduce the dimensionality and control the number of parameters between dense blocks.
- They typically consist of a batch normalization layer, followed by a 1x1 convolutional layer and a downsampling operation (e.g., average pooling).

3. Growth Rate:

- The growth rate, denoted as k, controls the number of feature maps produced by each layer within a dense block.
- Given an input feature map with mm channels, each layer within the dense block produces k new feature maps, resulting in a total of $\mathrm{m+k}$ output channels.



Mathematics:

- Let H_l represent the output feature map of the lth layer in a dense block, and H_0 be the input feature map.
- The output of each layer is computed as:
  $H_l = H_{l-1} \oplus \mathrm{Conv}_{3\times3}(\mathrm{BN}(H_{l-1}))$
  where $\oplus$ denotes concatenation, $\mathrm{Conv}_{3\times3}$ represents a 3×3 convolution, and BN denotes batch normalization.
- The growth rate k controls the number of output channels of each layer within the dense block.
- Transition layers reduce the dimensionality and spatial resolution of feature maps using pooling operations and 1×1 convolutions.

EfficientNet

EfficientNet is a convolutional neural network (CNN) architecture designed to achieve high accuracy while being computationally efficient, making it well-suited for resource-constrained environments such as mobile devices and edge devices. Here's a concise explanation of EfficientNet along with its mathematics:

1. Compound Scaling:

- EfficientNet uses a technique called compound scaling to simultaneously scale the network's depth, width, and resolution.
- It balances these dimensions to maximize model efficiency while maintaining high accuracy.

2. Depth, Width, and Resolution Scaling:

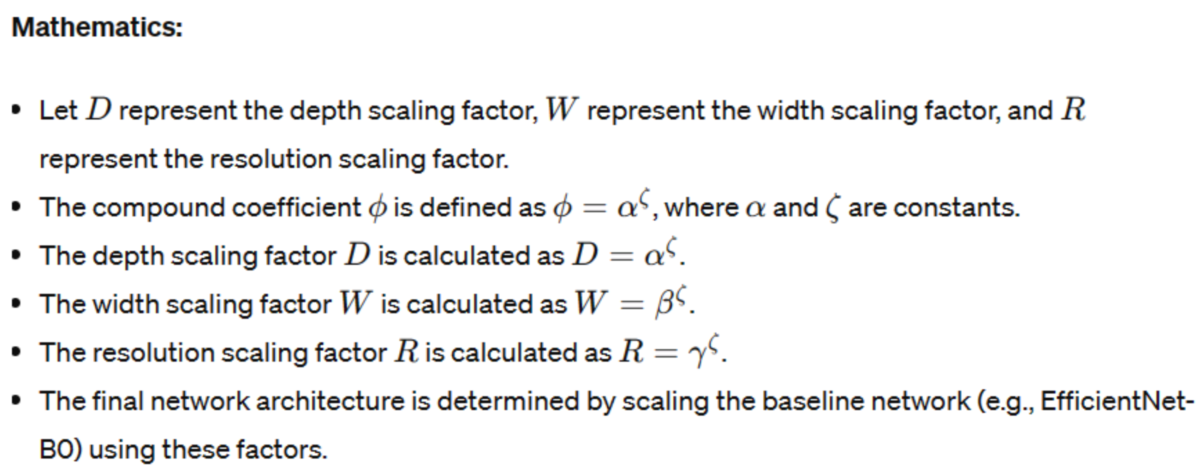
- Depth scaling refers to increasing the number of layers in the network.
- Width scaling involves increasing the number of channels (width) in each layer.
- Resolution scaling means increasing the input image resolution.

3. Efficient Blocks:

- EfficientNet utilizes a combination of efficient building blocks, including inverted residual blocks and linear bottleneck blocks.
- These blocks are designed to maximize representational power while minimizing computational cost.

4. Compound Coefficients:

- EfficientNet introduces compound coefficients (φ) to control the scaling of depth, width, and resolution.
- The compound coefficient (φ) determines how much to scale each dimension relative to the baseline network.



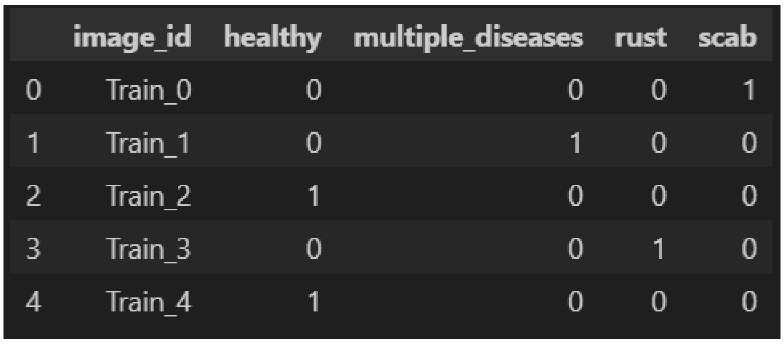
Mathematics:

- Let D represent the depth scaling factor, W represent the width scaling factor, and R represent the resolution scaling factor.
- The compound coefficient ϕ is defined as $\phi = \alpha^{\zeta}$, where α and ζ are constants.
- The depth scaling factor D is calculated as $D = \alpha^{\zeta}$.
- The width scaling factor W is calculated as $W = \beta^{\zeta}$.
- The resolution scaling factor R is calculated as $R = \gamma^{\zeta}$.
- The final network architecture is determined by scaling the baseline network (e.g., EfficientNet-B0) using these factors.

Explanation of the approach and results of the Project

1. Imports and Setup
First, we import the necessary libraries and modules such as OpenCV (cv2), NumPy (np), TensorFlow (tf), Keras, EfficientNet, Pandas (pd), Matplotlib (plt), Seaborn (sns), Plotly, Scikit-learn, and other utility libraries. We then set the random seed for reproducibility and filters out warnings.

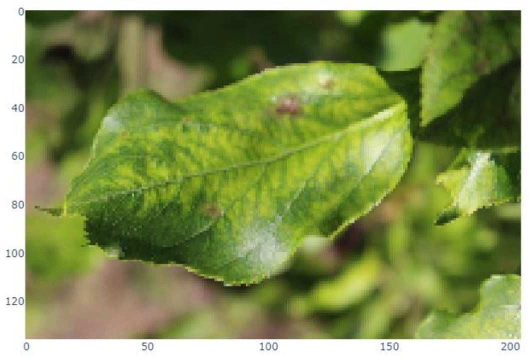
2. Data Loading and Exploration
The paths for the dataset is defined, including images, test data, train data, and sample submission files. We then load the train and test data into Pandas DataFrames and performs exploratory data analysis (EDA) on the dataset.



	image_id	healthy	multiple_diseases	rust	scab
0	Train_0	0	0	0	1
1	Train_1	0	1	0	0
2	Train_2	1	0	0	0
3	Train_3	0	0	1	0
4	Train_4	1	0	0	0

3. Exploratory Data Analysis (EDA)
Visualizing Sample Leaf Images
- Sample leaf images from each category are visualised, allowing for a visual understanding of the characteristics of healthy leaves and leaves affected by different diseases.
- Healthy leaves appear completely green without any brown/yellow spots or scars.

- Leaves with scab disease exhibit large brown marks and stains across the leaf surface, indicative of fungal or bacterial infections.
- Leaves with rust disease show several brownish-yellow spots across the leaf, caused by a specific type of rust fungi.
- Leaves with multiple diseases display symptoms of multiple diseases, including brown marks and yellow spots.



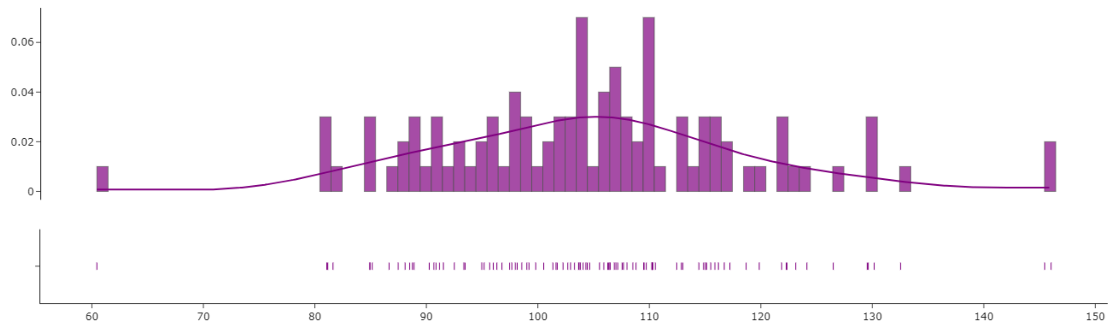
Sample visualisation

I have plotted the first image in the training data above (the RGB values can be seen by hovering over the image). The green parts of the image have very low blue values, but by contrast, the brown parts have high blue values. This suggests that green (healthy) parts of the image have low blue values, whereas unhealthy parts are more likely to have high blue values. This might suggest that the blue channel may be the key to detecting diseases in plants.

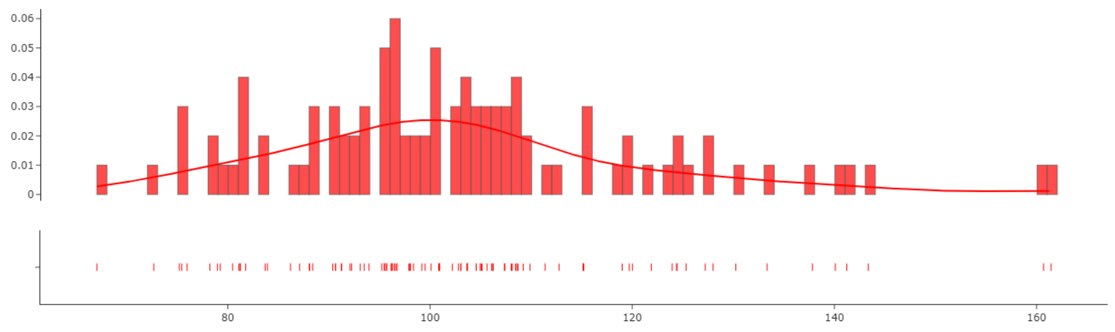
Channel Distributions

- We then analysed the distributions of red, green, and blue channel values across the dataset.
- It is observed that the green channel values have a higher magnitude compared to the red and blue channels, which is expected for leaf images as they are predominantly green.
- The green channel values have a more uniform distribution with a peak around 140, while the red channel values have a slight positive skew, and the blue channel values have a more uniform distribution with minimal skew.
- When plotted together, the channel value distributions appear to have a similar shape but are shifted horizontally.

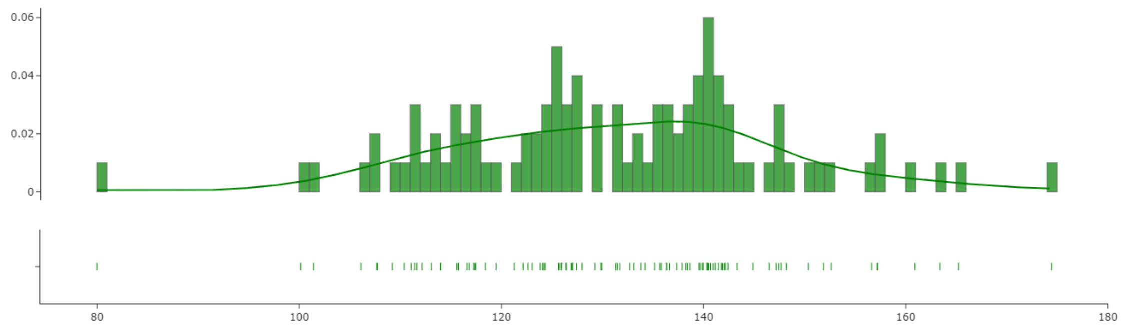
Distribution of channel values



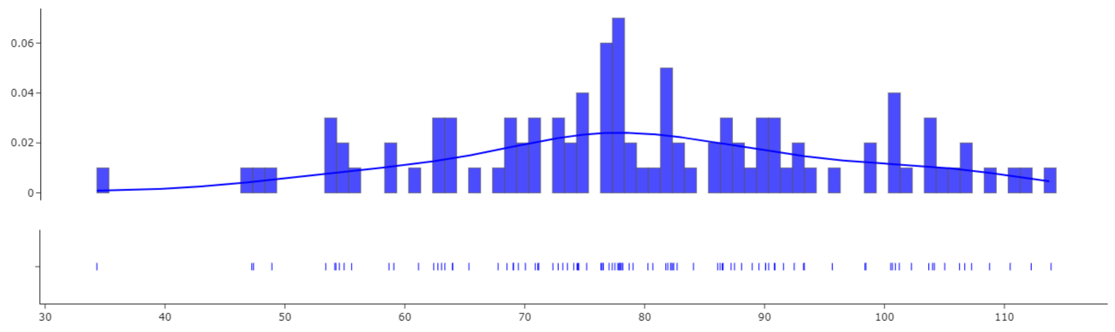
Distribution of red channel values

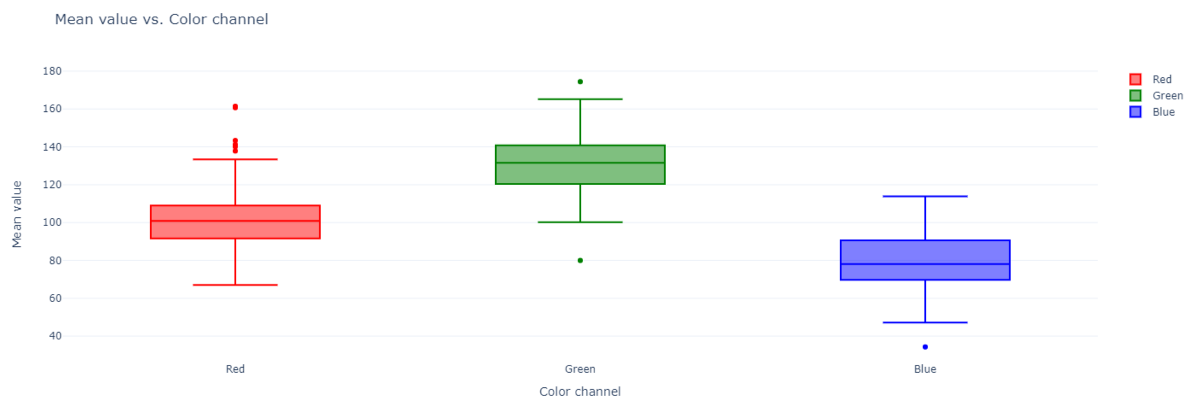


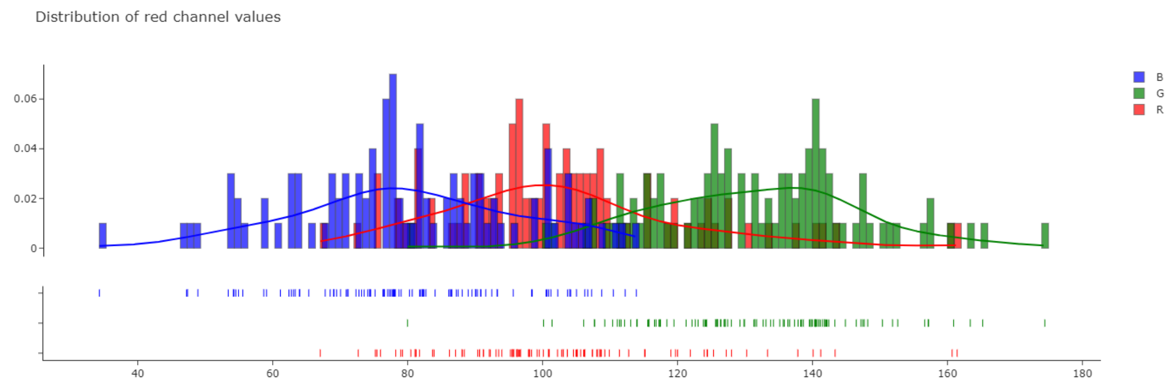
Distribution of green channel values



Distribution of blue channel values



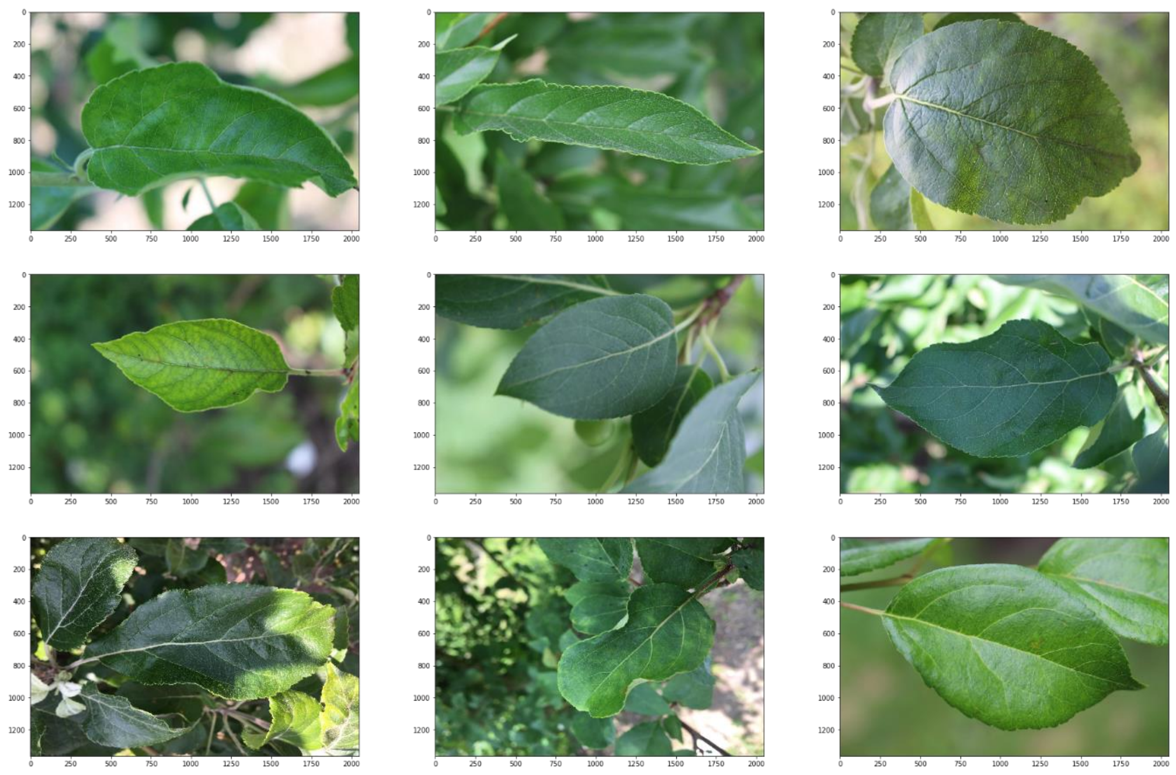




From the above plots, we can clearly see which colors are more common and which ones less common in the leaf images. Green is the most pronounced color, followed by red and blue respectively. The distributions, when plotted together, appear to have a similar shape, but shifted horizontally.

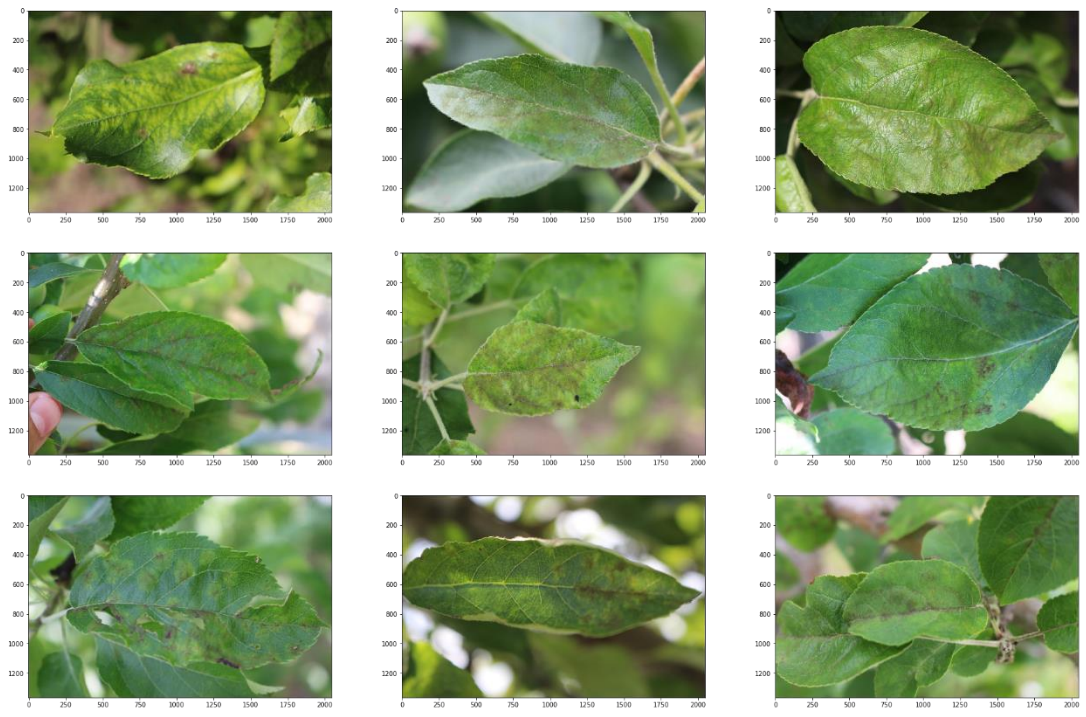
Sample leaves visualisation
- The `load_image` function is defined to load and preprocess individual images from the dataset.
- The code visualizes the distribution of channel values (red, green, blue) across the images using histograms and box plots created with Plotly's `ff.create_distplot` and `go.Box` functions.
- It provides a function `visualize_leaves` to display specific examples of healthy, scab, rust, and multiple disease-affected leaves from the dataset.
- The code creates parallel categories plots using `px.parallel_categories` to analyze the distribution of target variables.
- It generates a pie chart using `go.Pie` to visualize the proportions of different target variables.
- Histograms are plotted using `px.histogram` to show the distributions of binary target variables (e.g., healthy, scab, rust, multiple diseases).

Now, I will visualize sample leaves belonging to different categories in the dataset.



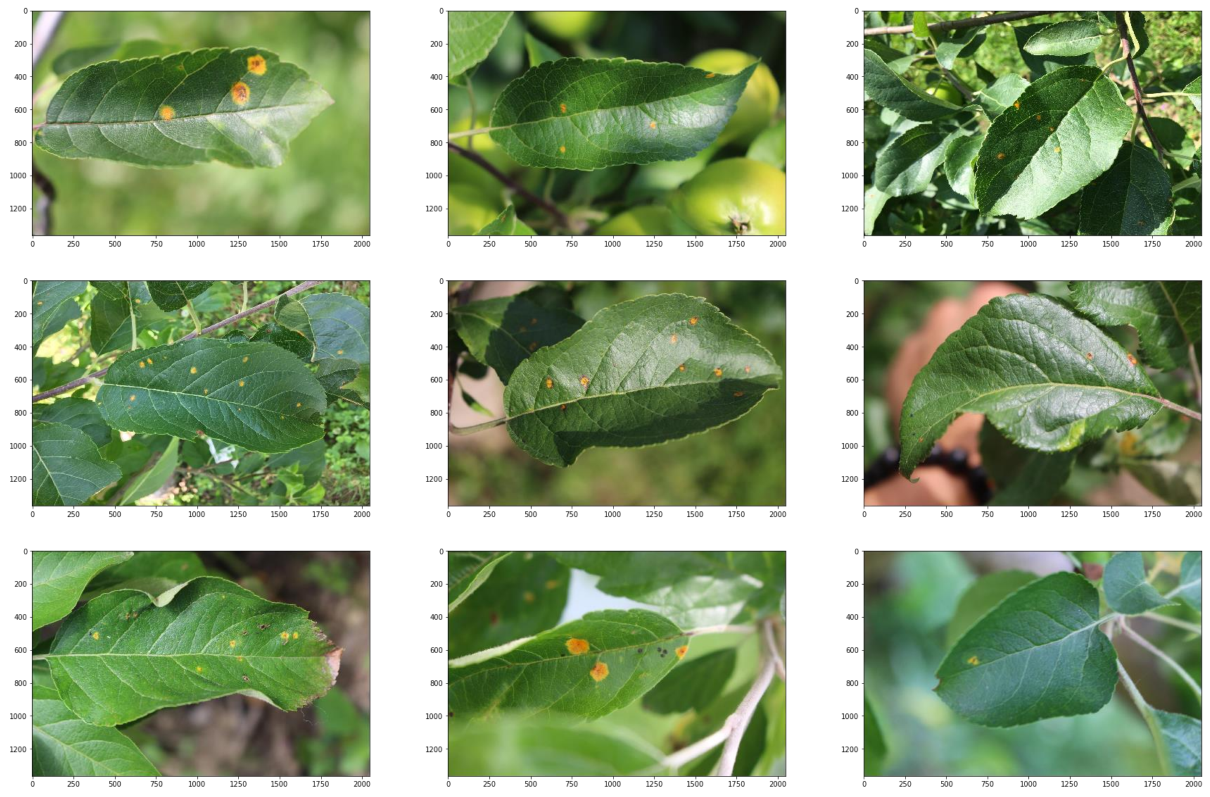
Healthy leaves

In the above images, we can see that the healthy leaves are completely green, do not have any brown/yellow spots or scars. Healthy leaves do not have scab or rust.



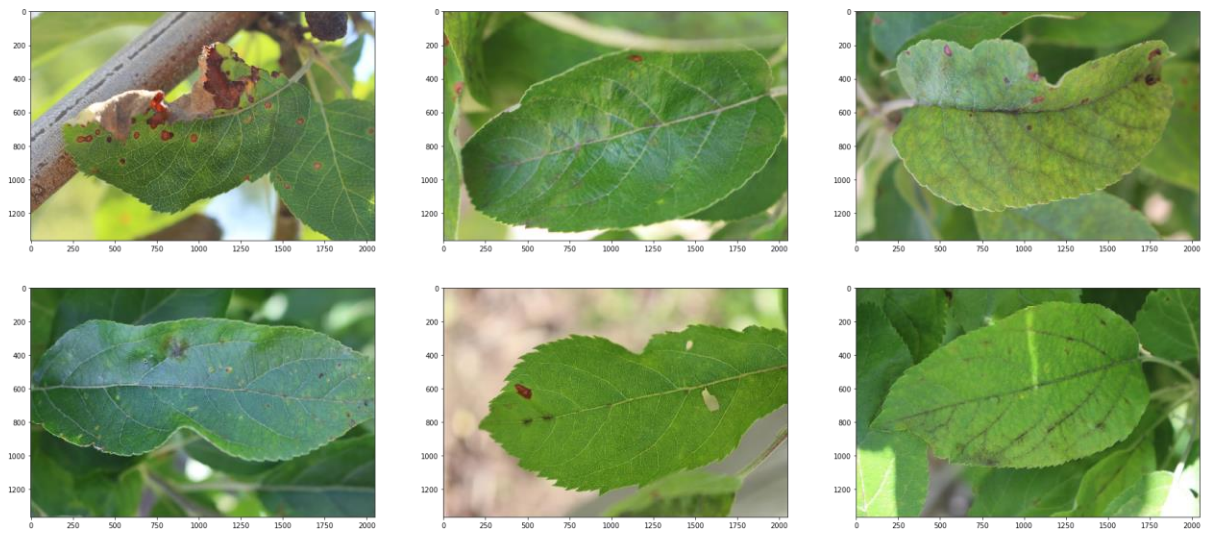
Leaves with scab

In the above images, we can see that leaves with "scab" have large brown marks and stains across the leaf. Scab is defined as "any of various plant diseases caused by fungi or bacteria and resulting in crustlike spots on fruit, leaves, or roots. The spots caused by such a disease". The brown marks across the leaf are a sign of these bacterial/fungal infections. Once diagnosed, scab can be treated using chemical or non-chemical methods.



Leaves with rust

In the above images, we can see that leaves with "rust" have several brownish-yellow spots across the leaf. Rust is defined as "a disease, especially of cereals and other grasses, characterized by rust-colored pustules of spores on the affected leaf blades and sheaths and caused by any of several rust fungi". The yellow spots are a sign of infection by a special type of fungi called "rust fungi". Rust can also be treated with several chemical and non-chemical methods once diagnosed.



Leaves with multiple diseases

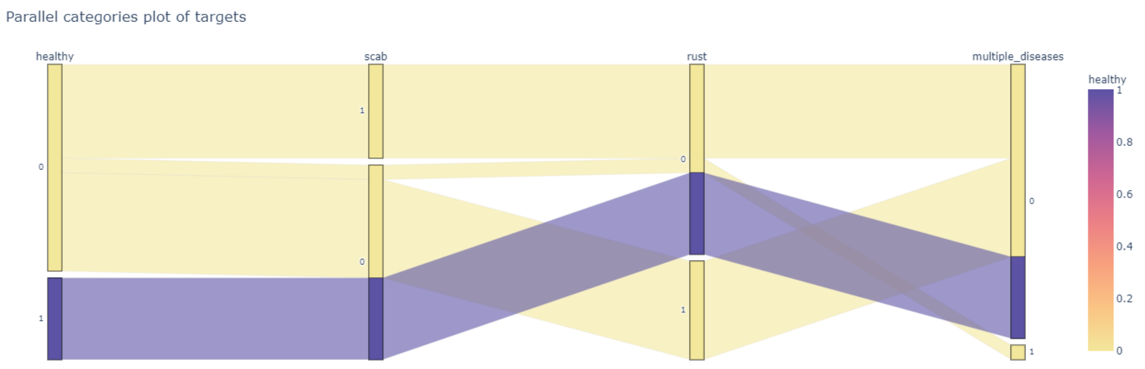
In the above images, we can see that the leaves show symptoms for several diseases, including brown marks and yellow spots. These plants have more than one of the above-described diseases.

Visualise Targets

Now, I will visualize the labels and target data. In all the below plots, blue represents the "desired" or "healthy" condition, and red represents the "undesired" or "unhealthy" condition.

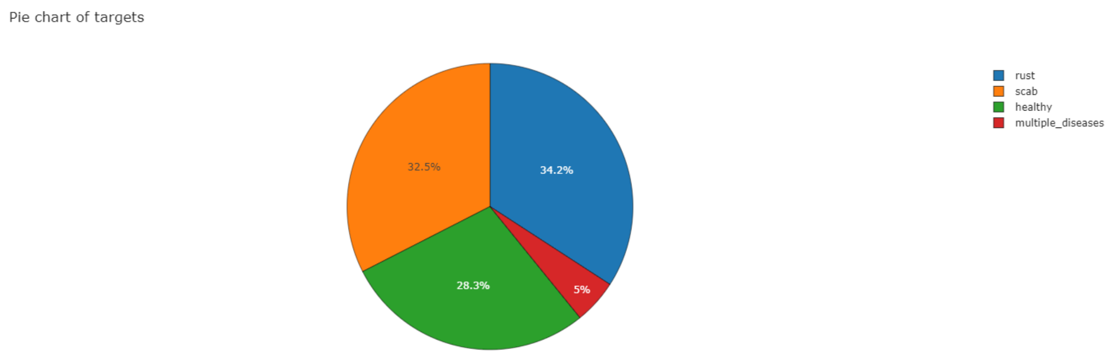
Parallel Plot (all labels together)

In the below plot, we can see the relationship between all four categories. As expected, it is impossible for a healthy leaf (healthy == 1) to have scab, rust, or multiple diseases. Also, every unhealthy leaf has one of either scab, rust, or multiple diseases. The frequency of each combination can be seen by hovering over the plot.



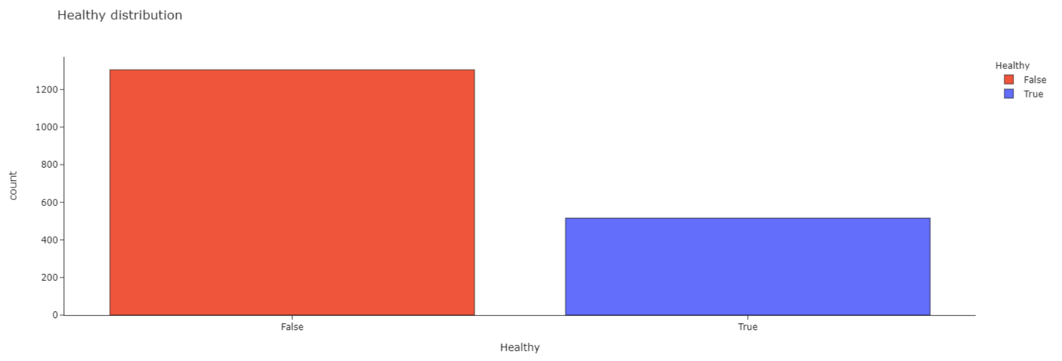
Pie Chart

Pie chart reveals that most leaves in the dataset (71.7%) are unhealthy, while approximately one-third of the leaves have scab or rust, and a small percentage (5%) have multiple diseases.



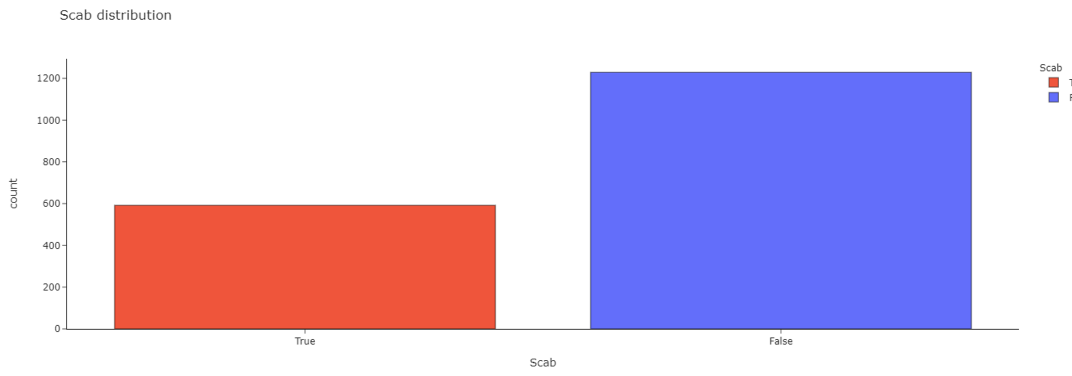
Healthy Distribution

We can see that there are more unhealthy (healthy == 0) plants than healthy (healthy == 1) ones. There are 1305 (72%) unhealthy plants and 516 (28%) healthy plants.



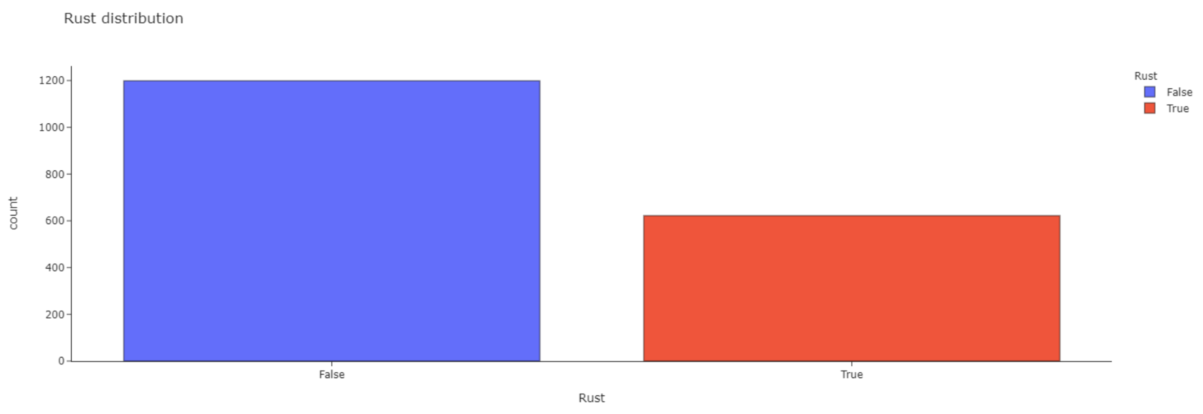
Distributions of individual target variables are also visualized.

Scab Distribution



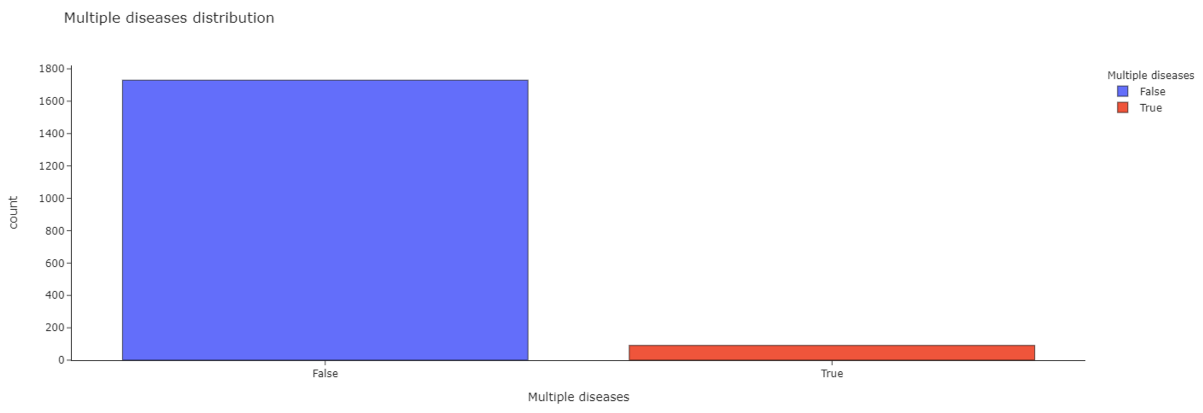
We can see that there are more plants without scab (scab == 0) than those with scab (scab == 1). There are 592 (33%) unhealthy plants and 1229 (67%) healthy plants.

Rust Distribution



We can see that there are more plants without rust (rust == 0) than those with rust (rust == 1). There are 622 (34%) unhealthy plants and 1199 (66%) healthy plants. We can see that the "unhealthy" percentage is very similar for both rust and scab.

Multiple diseases Distribution

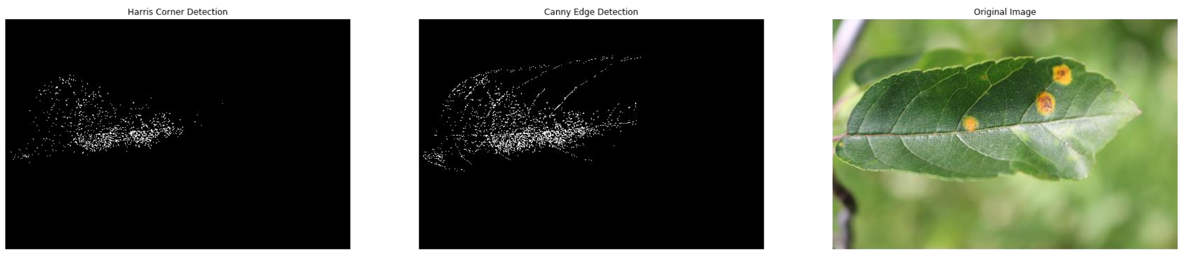


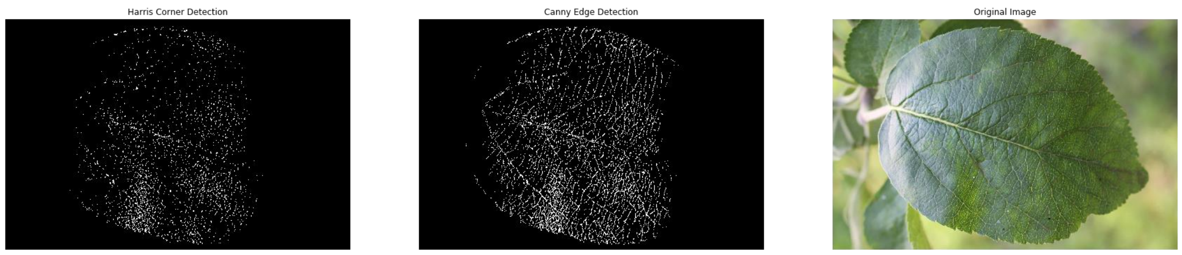
We can see that very few leaves have multiple diseases (it is a rare occurrence). There are 91 (5%) unhealthy plants and 1730 (95%) healthy plants.

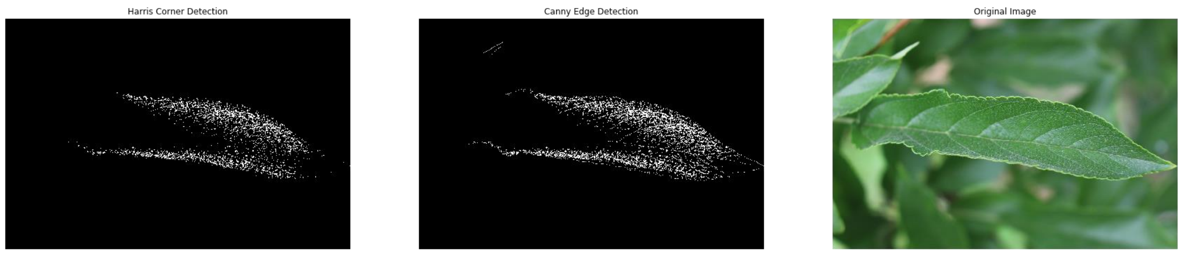
4. Image Processing and Augmentation

Canny Edge Detection

The Canny edge detection algorithm is applied to the leaf images to detect the edges of objects present in the images.
The `detect_corners_and_edges` function uses Harris Corner Detection (`cv2.cornerHarris`) and Canny Edge Detection (`cv2.Canny`) to identify corners and edges in images, respectively. The results are visualized using Matplotlib.



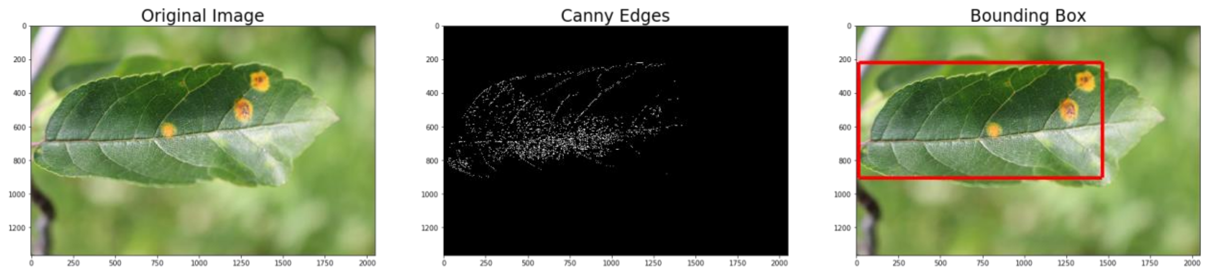


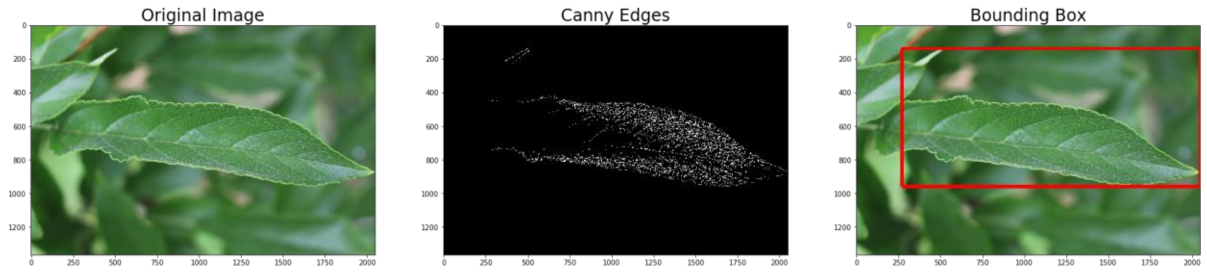


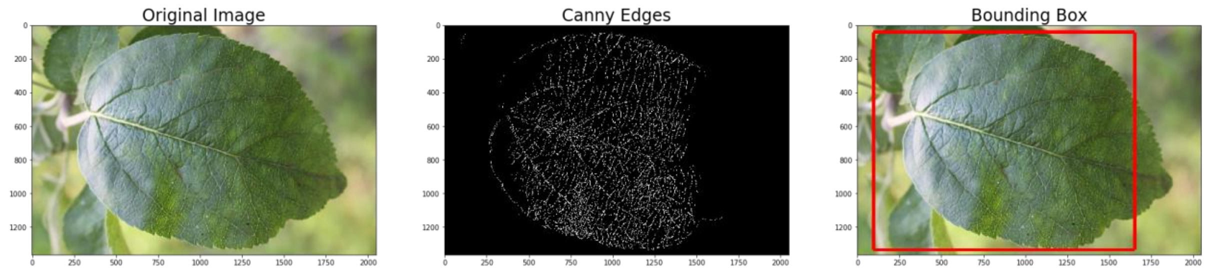
Edge Detection and Bounding Box

Detected edges can be used to predict a bounding box around the leaf by identifying the most extreme edges at the four corners of the image.

The `edge_and_cut` function performs edge detection using Canny Edge Detection, calculates the bounding box coordinates based on the edge coordinates, and displays the original image, edge image, and bounding box visualization.



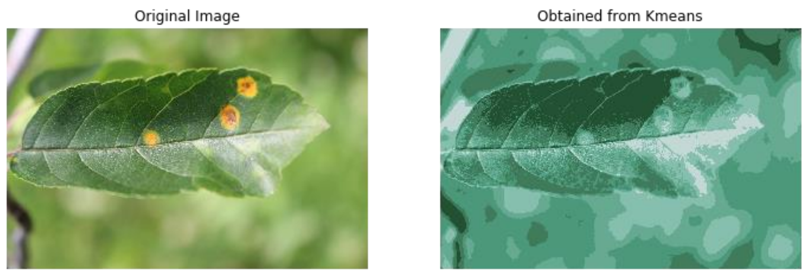




The second column of images above contains the Canny edges and the third column contains cropped images. I have taken the Canny edges and used it to predict a bounding box in which the actual leaf is contained. The most extreme edges at the four corners of the image are the vertices of the bounding box. This red box is likely to contain most of it but not all of the leaf. These edges and bounding boxes can be used to build more accurate models.

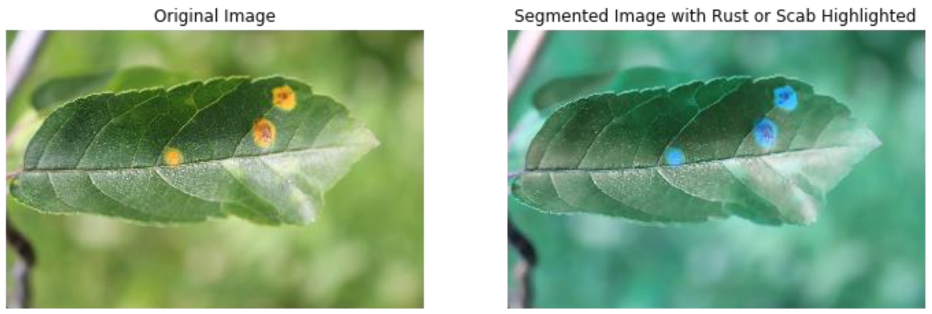
K-Means Clustering

The `kmeans_clusters` function applies K-Means clustering (`sklearn.cluster.KMeans`) to segment images based on colour. The segmented image is displayed alongside the original image.



Mean Shift Segmentation

The `mean_shift_segmentation` function uses the Mean Shift algorithm (`cv2.pyrMeanShiftFiltering`) for image segmentation. It highlights potential regions of rust or scab on the leaves by drawing circles around the contours.



Flipping

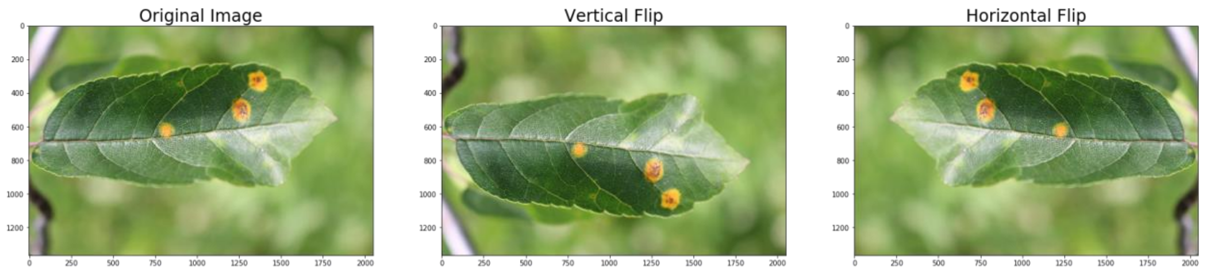
Flipping is a simple transformation that involves index-switching on the image channels. In vertical flipping, the order of rows is exchanged, whereas in vertical flipping, the order of rows is exchanged. Let us assume that A_{ijk} (of size (m, n, 3)) is the image we want to flip. Horizontal and vertical flipping can be represented by the transformations below:

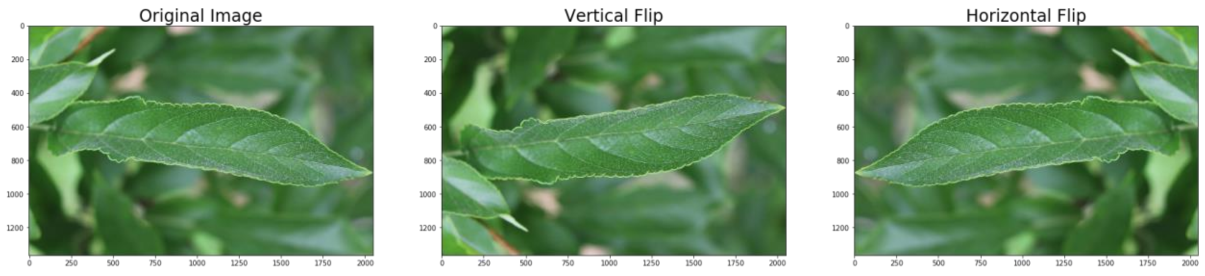
$$Image = A_{ijk}$$

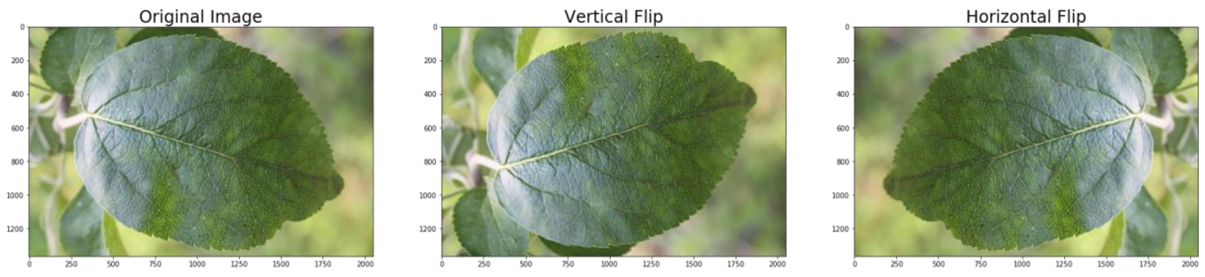
$$Horizontal flip : A_{ijk} \rightarrow A_{i(n+1-j)k}$$

$$Vertical flip : A_{ijk} \rightarrow A_{(m+1-i)jk}$$

We can see that the order of columns is exchanged in horizontal flipping. While the i and k indices remain the same, the j index reverses. Whereas, in vertical flipping, the order of rows is exchanged in horizontal flipping. While the j and k indices remain the same, the i index reverses.





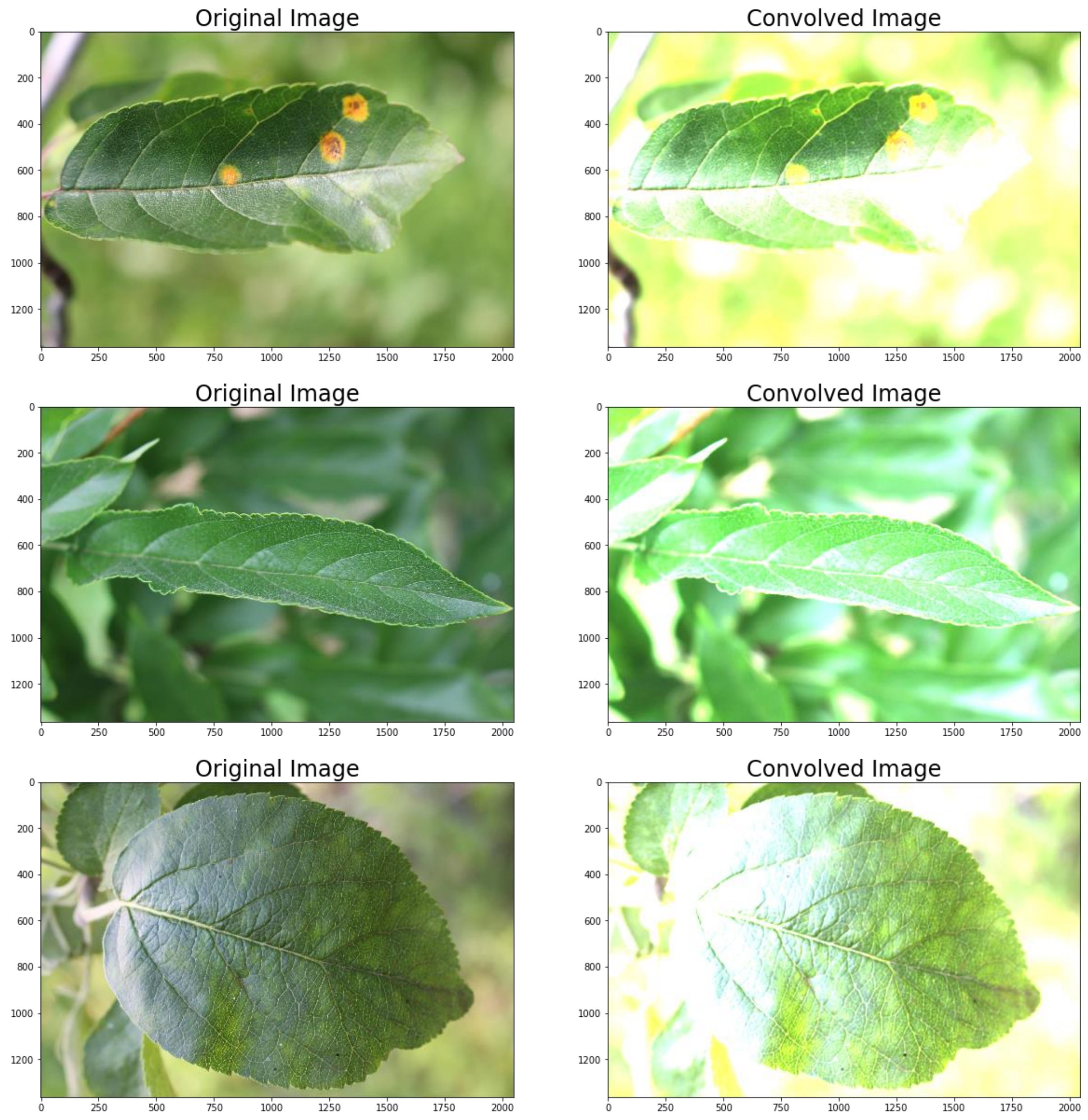


The `invert` function demonstrates image flipping (vertical and horizontal) using `cv2.flip`.

We can see that the images are simply flipped. All major features in the image remain the same, but to a computer algorithm, the flipped images look completely different. These transformations can be used for data augmentation, making models more robust and accurate.

Convolution

Convolution is a rather simple algorithm which involves a kernel (a 2D matrix) which moves over the entire image. The `conv` function applies a convolution operation (`cv2.filter2D`) to the input image.



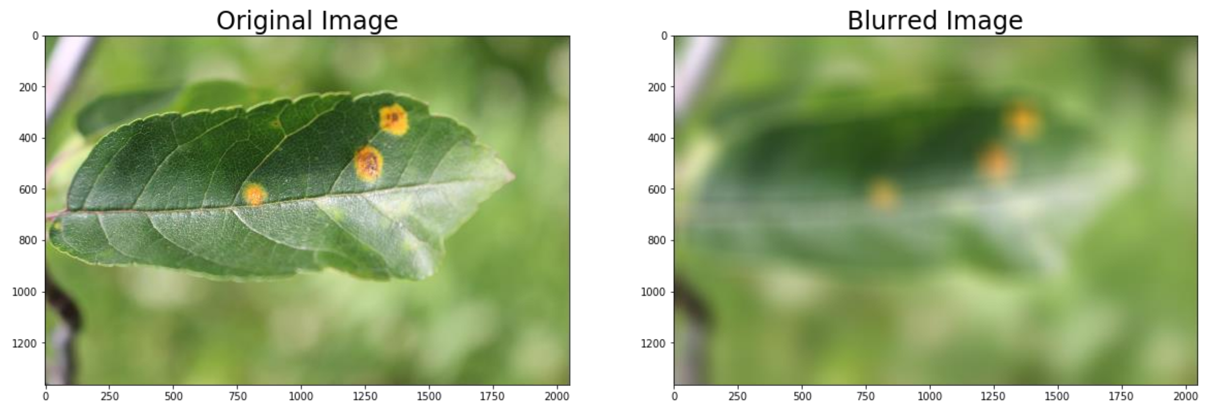
The convolution operator seems to have an apparent "sunshine" effect of the images. This serves the purpose of augmenting the data, thus helping to build more robust and accurate models.

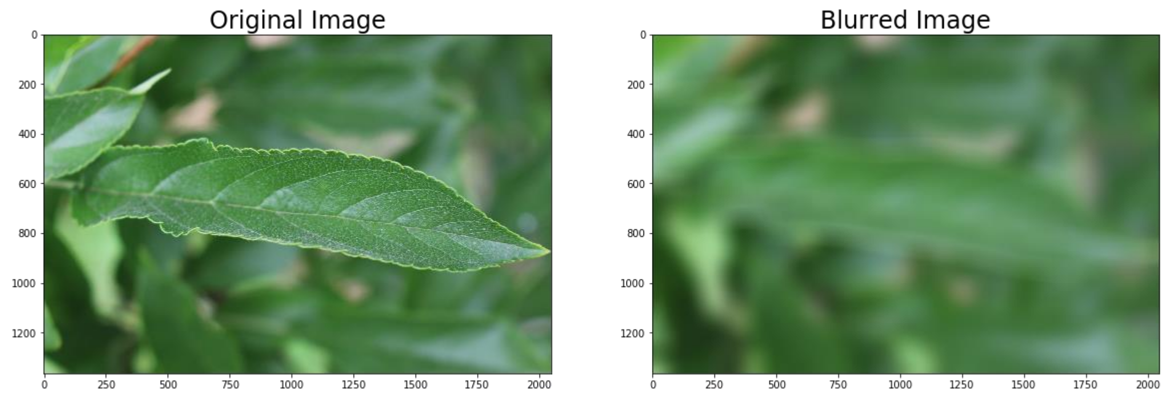
Blurring

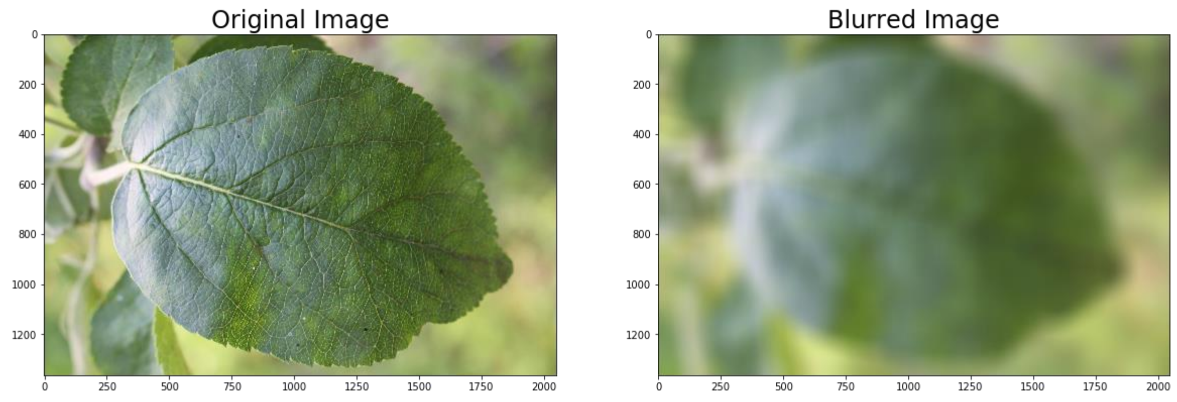
Blurring is simply the addition of noise to the image, resulting in a less-clear image. The noise can be sampled from any distribution of choice, as long as the main content in the image does not become invisible. Only the minor details get obfuscated due to blurring. The blurring transformation can be represented using the equation below.

$$A_{ijk} = A_{ijk} + \mathcal{N}(0, 0.1)$$

The `blur` function applies a blurring effect to the input image using `cv2.blur`. We uses a Gaussian distribution with mean 0 and variance 0.1. Below I demonstrate the effect of blurring on a few leaf images:







The transformation clearly blurs the image by removing detailed, low-level features, while retaining the major, high-level features. This is once again a great way to augment images and train more robust models.

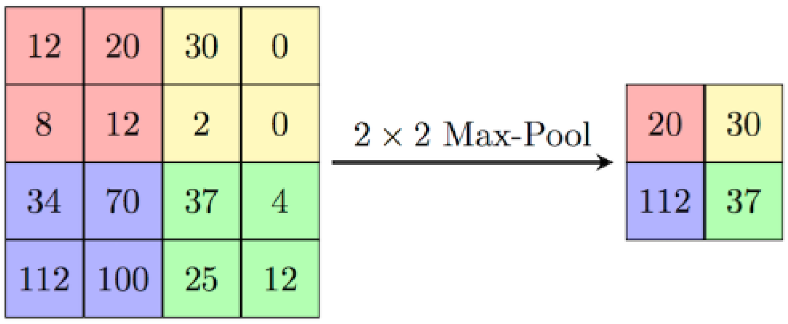
5. Modelling

Preparing the ground

Before we move on to building the models, I will explain the major building blocks in pretrained CV models. Every major ImageNet model has a different architecture, but each one has the common building blocks: **Conv2D, MaxPool, ReLU**. I have already explained the mechanism behind convolution in the previous section, so I will now explain MaxPool and ReLU.

MaxPool

Max pooling is very similar to convolution, except it involves finding the maximum value in a window instead of finding the dot product of the window with a kernel. Max pooling does not require a kernel and it is very useful in reducing the dimensionality of convolutional feature maps in CNNs. The image below demonstrates the working of MaxPool:

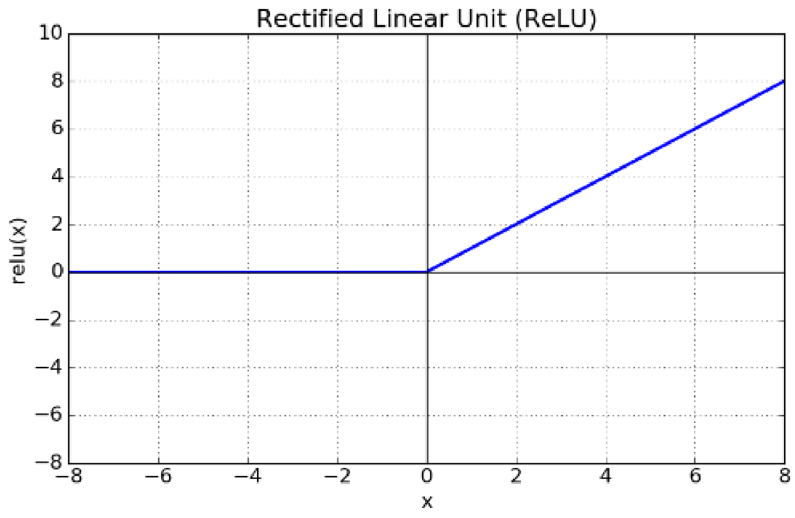


The above example demonstrates max pooling with a window size of *(2, 2)*. This process can be represented with the equation below:

$$Maxpool(f,h)_{mn} = max(m+j-1, n+k-1 \ \forall \ 1 \leq m \leq l, \ 1 \leq n \leq l)$$

In the above equation, the window moves across the image and the maximum value in each winow is calculated. Once again, this process is very important in reducing the complexity of CNNs while retaining features.

ReLU
ReLU is an activation function commonly used in neural network architectures. *ReLU(x)* returns 0 for *x < 0* and *x* otherwise. This function helps introducenon-linearity in the neural network, thus increasing its capacity ot model the image data. The graph and equation of *ReLU* are:



$$G_{mn} = ReLU(x)_{mn} = \begin{cases} 0 & \text{if } x_{mn} < 0 \\ x_{mn} & \text{if } x_{mn} \geq 0 \end{cases}$$

As mentioned earlier, this function is non-linear and helps increase the modeling capacity of the CNN models. Now since we understand the basic building blocks of pretrained images models, let us finetune some pretained ImageNet models on TPU and visualize the results!

6. Model Building and Training
We build and train three different deep learning models:

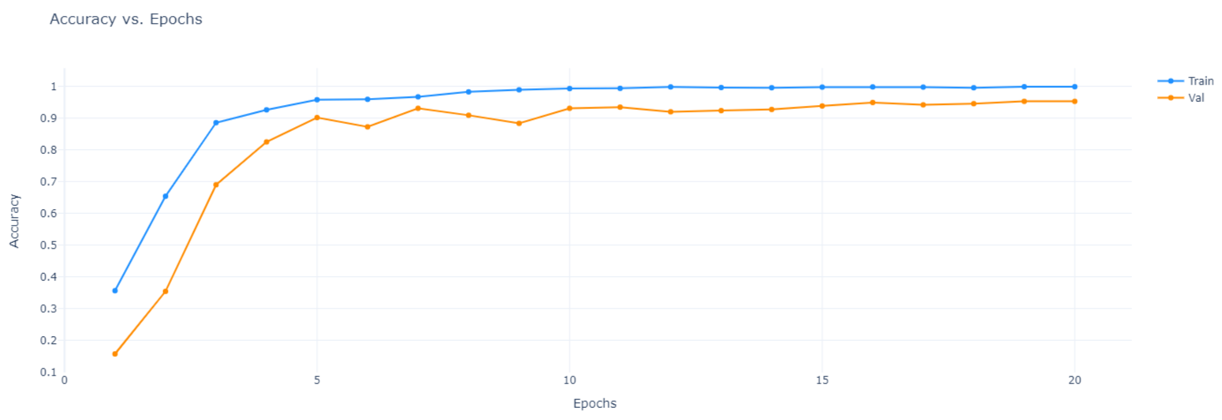
DenseNet121
The DenseNet model is trained on the leaf image dataset, and its performance is visualized using scatter plots, animations, and sample predictions.
DenseNet predicts leaf diseases with great accuracy, and the probabilities are highly polarized, indicating that the model makes predictions with high confidence.

- A pre-trained DenseNet121 model from `tf.keras.applications.DenseNet121` is used as the base model.
- The model is constructed by adding a global average pooling layer (`L.GlobalAveragePooling2D`) and a dense layer (`L.Dense`) with a softmax activation for multi-class classification.

- The model is compiled with the 'adam' optimizer and categorical cross-entropy loss.
- The model architecture is visualized using `tf.keras.utils.model_to_dot` and rendered as an SVG image.
- The model is trained for a specified number of epochs (`EPOCHS`) using the `model.fit` function, with a learning rate schedule (`lr_schedule`) defined by the `build_lrfn` function.
- Training and validation curves for accuracy are plotted using Plotly's `go.Scatter`.

Now let us train DenseNet on leaf images and evaluate its performance.

Result Visualisation

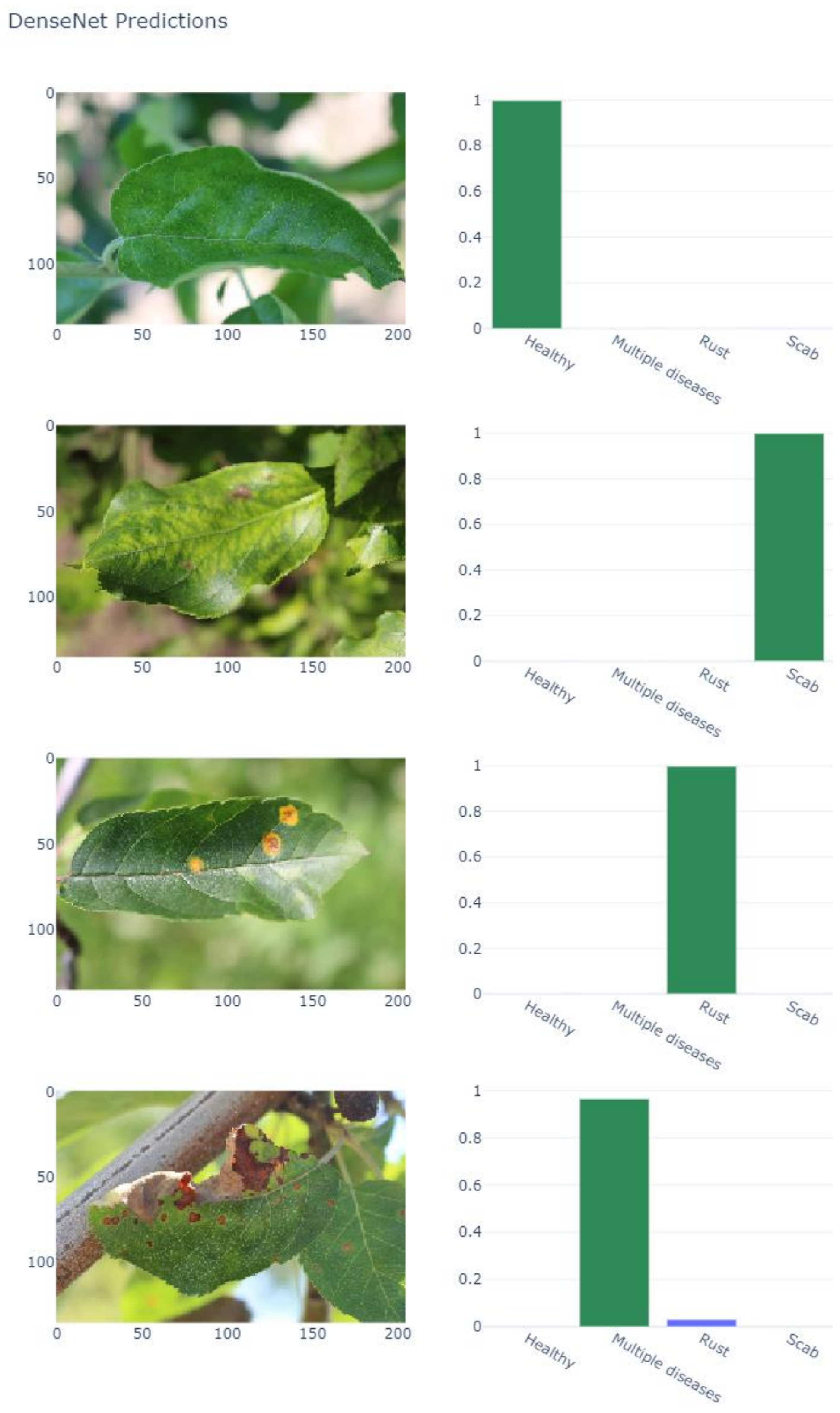


Scatter plot

From the above plots, we can see that the losses decrease and accuracies increase quite consistently. The training metrics settle down very fast (after 1 or 2 epochs), whereas the validation metrics much greater volatility and start to settle down only after 7-8 epochs. This is expected because validation data is unseen and more diffcult to make predictions on than training data.

Sample Predictions

Now, I will visualize some sample predictions made by the DenseNet model. The "red" bars represent the model's prediction (maximum probability), the "green" bars represent the ground truth (label), and the rest of the bars are "blue”. When the model predicts correctly, the prediction bar is "green".



We can see that DenseNet predicts leaf diseases with great accuracy. No red or blue bars are seen. The probabilities are very polarized (one very high and the rest very low), indicating that the model is making these predictions with great confidence.

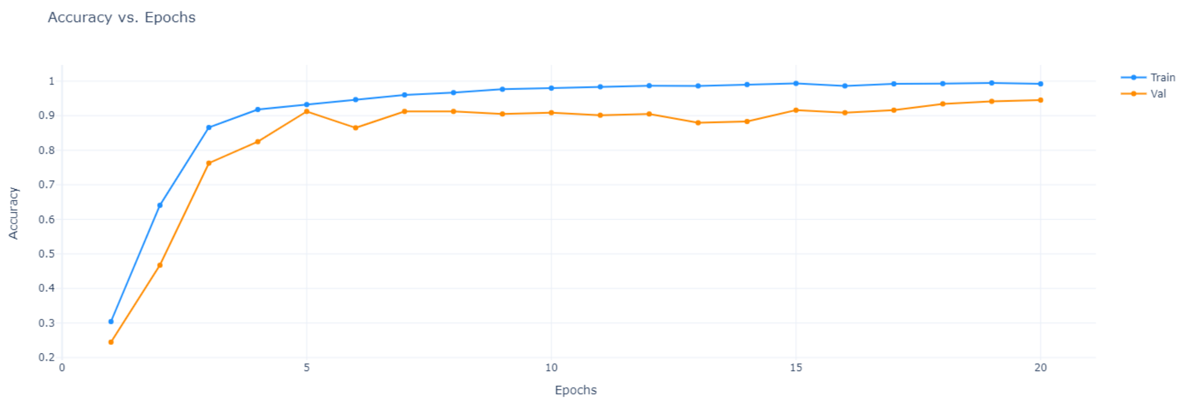
EfficientNetB7

EfficientNet performs model scaling in an innovative way to achieve excellent accuracy with significantly fewer parameters compared to architectures like ResNet and DenseNet.
The fundamental building block of the EfficientNet architecture involves more addition and multiplication-based operators than concatenation, which is more common in DenseNet.
The EfficientNet model is trained on the leaf image dataset, and its performance is visualized and analyzed.
The performance of EfficientNet to DenseNet is compared, noting that EfficientNet does not exhibit high volatility in validation metrics compared to DenseNet.

- An EfficientNetB7 model from the `efficientnet.tfkeras` library is used, with weights initialized from ImageNet.
- Similar to the DenseNet121 model, it is constructed by adding a global average pooling layer and a dense layer with a softmax activation.
- The model is compiled, trained, and evaluated in a similar manner to the DenseNet121 model.

Now let us train EfficientNet on leaf images and evaluate its performance.

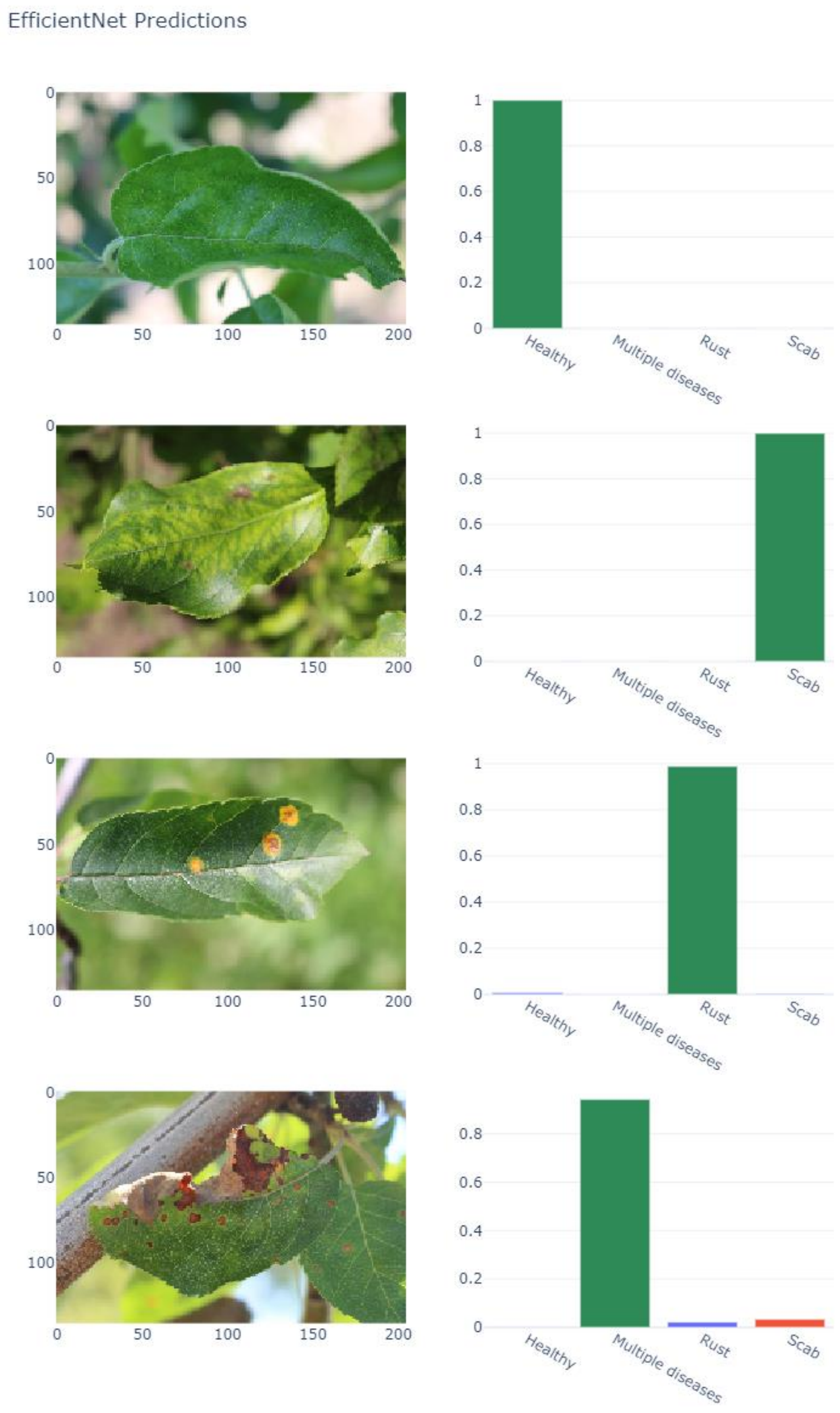
Result Visualisation



Scatter plot

Sample Predictions

Now, I will visualize some sample predictions made by the EfficientNet model. The "red" bars represent the model's prediction (maximum probability), the "green" bars represent the ground truth (label), and the rest of the bars are "blue”. When the model predicts correctly, the prediction bar is "green".



The model predicts the leaf diseases with great accuracy. The level of performance is similar to that of DenseNet, as the green bars are very common. The red and blue bars are more prominent in the last (fourth) leaf labeled "multiple diseases". This is probably because leaves with multiple diseases may show symptoms of rust and scab as well, thus slightly confusing the model.

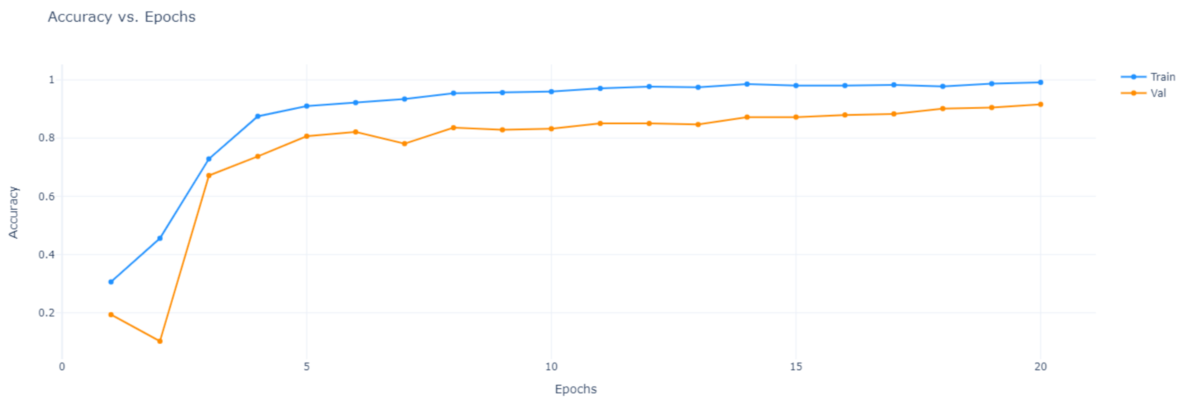
EfficientNetB7 with Noisy Student Weights

This model uses semi-supervised learning on noisy images to learn rich visual representations.
EfficientNet NoisyStudent outperformed the base EfficientNet on several tasks.
The EfficientNet NoisyStudent model is trained on the leaf image dataset, and its performance is visualized and compared to the base EfficientNet model.
Semi-supervised weights of EfficientNet NoisyStudent seem to set it apart from the base EfficientNet, as it exhibits better performance on the leaf disease classification task.

- It is another EfficientNetB7 model, but with weights initialized from the Noisy Student pre-training.
- The model construction, training, and evaluation process is similar to the previous models.

Now let us train EfficientNet NoisyStudent on leaf images and evaluate its performance.

Result Visualisation

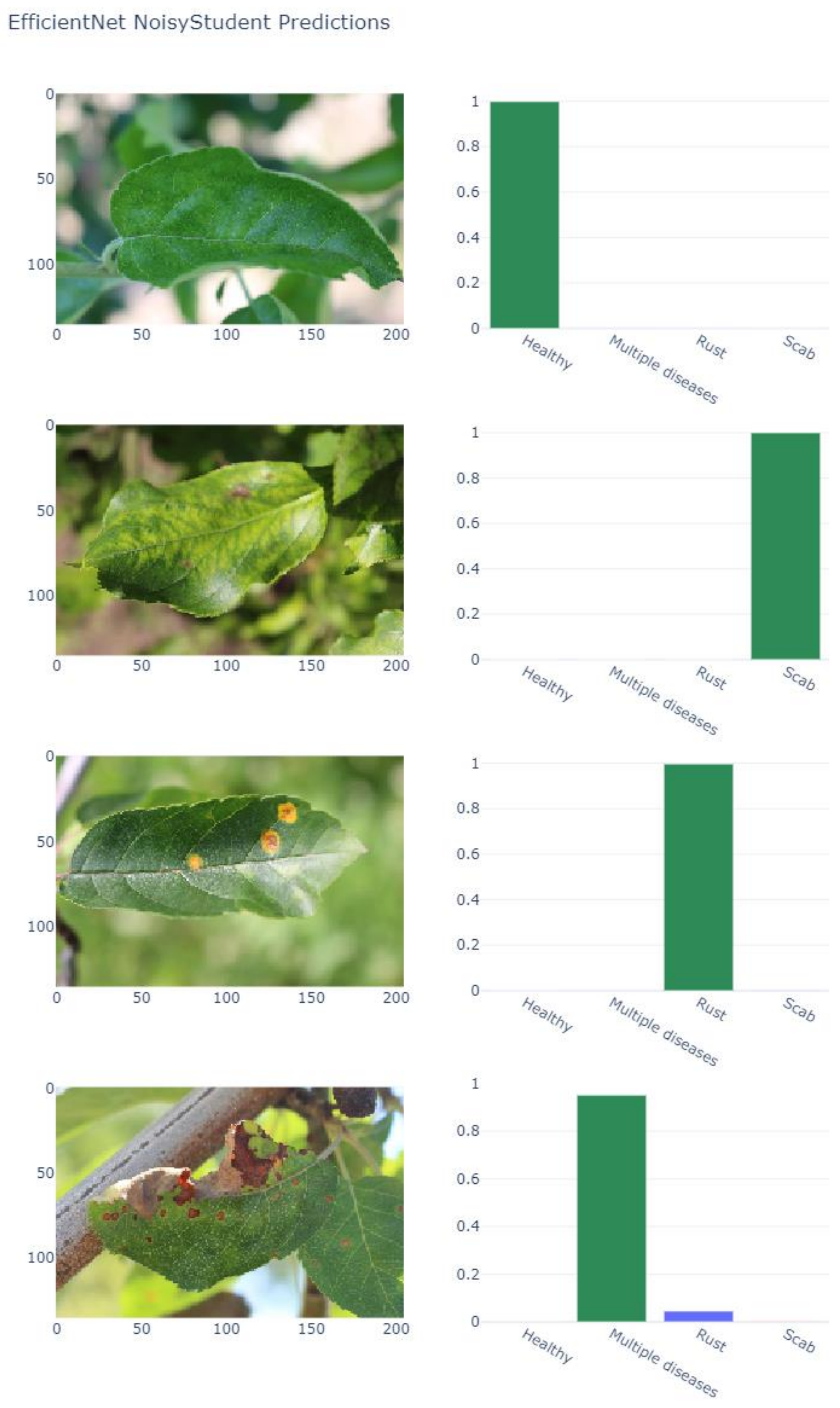


Scatter plot

From the above plots, we can see that the losses decrease and accuracies increase quite consistently. The training metrics settle down very fast (after 1 or 2 epochs), whereas the validation metrics much greater volatility and start to settle down only after 12-13 epochs (similar to DenseNet). This is expected because validation data is unseen and more diffcult to make predictions on than training data.

Sample Predictions

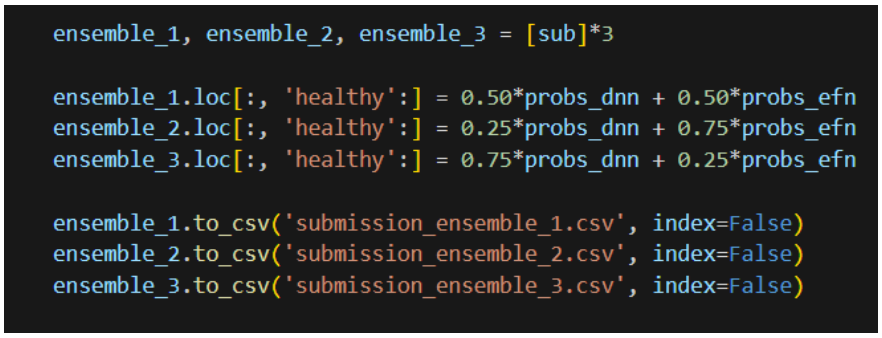
Now, I will visualize some sample predictions made by the EfficientNet NoisyStudent model. The "red" bars represent the model's prediction (maximum probability), the "green" bars represent the ground truth (label), and the rest of the bars are "blue”. When the model predicts correctly, the prediction bar is "green".



Similar to the DenseNet model, EfficientNet NoisyStudent predicts leaf diseases with great accuracy. No red bars are seen. The probabilities are very polarized (one very high and the rest very low), indicating that the model is making these predictions with great confidence. The semi-supervised weights seem to set this model apart from EfficientNet.
The red and blue bars are, once again, more prominent in the last (fourth) leaf labeled "multiple_diseases". This is probably because leaves with multiple diseases may show symptoms of rust and scab as well, thus slightly confusing the model.

7. Ensembling

Ensembling involves the averaging of multiple prediction vectors to reduce errors and improve accuracy.
Now, I will ensemble predictions from DenseNet and EfficientNet to (hopefully) produce better results.



```
ensemble_1, ensemble_2, ensemble_3 = [sub]*3

ensemble_1.loc[:, 'healthy':] = 0.50*probs_dnn + 0.50*probs_efn
ensemble_2.loc[:, 'healthy':] = 0.25*probs_dnn + 0.75*probs_efn
ensemble_3.loc[:, 'healthy':] = 0.75*probs_dnn + 0.25*probs_efn

ensemble_1.to_csv('submission_ensemble_1.csv', index=False)
ensemble_2.to_csv('submission_ensemble_2.csv', index=False)
ensemble_3.to_csv('submission_ensemble_3.csv', index=False)
```

- Ensemble submissions are created by combining the predictions from different models with varying weightages.
- Three ensemble submissions are generated: `submission_ensemble_1.csv`, `submission_ensemble_2.csv`, and `submission_ensemble_3.csv`.
- Each ensemble submission is created by taking a weighted average of the predictions from the DenseNet121 and EfficientNetB7 models, with different weightages for each ensemble.

8. Conclusion

Key findings and contributions of this project are:

- Image processing and augmentation techniques, such as edge detection, depth estimation, flipping, convolution, and blurring, can be used to preprocess the data and potentially improve model performance.
- Pre-trained CNN models like DenseNet, EfficientNet, and their variants can achieve high accuracy in classifying leaf diseases, with the EfficientNet NoisyStudent model exhibiting superior performance due to its semi-supervised training approach.
- Ensembling, stacking, and strong validation techniques may lead to more accurate and robust models for the plant disease classification task.

Overall, this detailed report provides a comprehensive understanding of the techniques and approaches employed in the plant disease classification.

9. References

1. Plant Pathology 2020 - FGVC7
2. [OpenCV Docs ~ by OpenCV] (https://docs.opencv.org/master/)
3. [Plant Pathology: Very Concise TPU EfficientNet ~ by xhulu] (https://www.kaggle.com/xhlulu/plant-pathology-very-concise-tpu-efficientnet)
4. [Plotly Express in Python ~ by Plotly] (https://plot.ly/python/plotly-express/)