Configuration Manual

MSc Research Project
Data Analytics

Aniket Shetty
Student ID: x23177861

School of Computing
National College of Ireland

Supervisor: Prof. Aaloka Anant

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Aniket Shetty
Student ID:	x23177861
Programme:	MSc. Data Analytics **Year:** Jan 2024-Jan 2025
Module:	MSc. Research Project
Lecturer:	Prof. Aaloka Anant
Submission Due Date:	12/12/2024
Project Title:	Corn Leaf Disease Detection Using Deep Learning and Explainable AI.
Word Count:	965 **Page Count:** 7

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Aniket Shetty
Date:	09/12/2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Aniket Shetty
X23177861

1 Introduction

This Research Project Aims at developing a Deep learning model using Resnet-50 and MobileNetV2 Architectures. Additionally, to get away from the problems of Black box Nature of the these models the author has developed Explainable AI Techniques like Grad CAM++ that will explain the reasons behind the models predictions. All the possible replication-related procedures are enumerated in this setup guide. A justification of the flow of project design from data collecting to model assessment. As necessary, Model implementation and code examples from several parts have also been included.

2 System Requirements

Hardware: GPU (e.g., NVIDIA Tesla T4)
Software: Python 3.x
Libraries: torch, torchvision, matplotlib, numpy, lime, pytorch-grad-cam, Pillow, scikit-learn,TensorFlow, cv2, scikit-learn.

Installation command for library

```
!pip install torch torchvision lime pytorch-grad-cam
!pip install tf-keras-vis
```

Platform: Google Colab or local environment supported by GPUs.

Research have made a use of Python 3.11.5 as the Programming language for local executions to build the model. Since Google Collab (GPU with 16 GB VRAM) provided faster execution and has a convenient and easily accessible environment for creating a Python notebook, this was used to create the environment necessary for the execution of the corn disease classification models. Google Collab having ready to use libraries from Google, the data was stored in Google Drive hence organization of data and code execution plus the means of storing the results were already organized. Simplicity of this system environment, and the scalability and capabilities to incorporate GPU for faster model training contributed to guiding the design of this.

3 Dataset Preparation

- **Download the dataset zip files from the below link.**

https://drive.google.com/drive/folders/196zmDkZqbs5LJQ8DqG0ZRkJC2va0UHP-?usp=sharing

The Dataset of corn leaf was originally taken from Kaggle which is an open-source platform. The Dataset was stored on the Google drive in our case and below is the code to mount Google Drive and access dataset files stored in it.

```
## Conection with Drive
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive;
emount=True).
```

Fig 1. Mount to Google Drive.

```
## Load and extract the dataset
input_path = "/content/drive/MyDrive/dataset.zip"

local_zip = input_path
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('/content/drive/MyDrive/')
zip_ref.close()
```

Fig 2. Unzip the dataset.zip file

- **If the dataset is zipped use the below python code to unzip it in the desired location.**

Load and extract the dataset
input_path = "/content/drive/MyDrive/dataset.zip". # Path to your zipped folder

local_zip = input_path
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('/content/drive/MyDrive/'). # Path to your Destination folder to unzip
zip_ref.close()

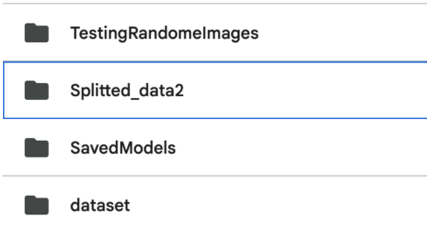

Fig 2.1 Datasets & Important Folders

Make sure you have these datasets saved and folders created(for example. SavedModels) as shown in Fig 2.1. located in your environment or google drive before execution of the codes.

3.1 Data Imbalance

Since there was class imbalance present in the dataset there was a need to manage this imbalanced class to avoid biased predictions.

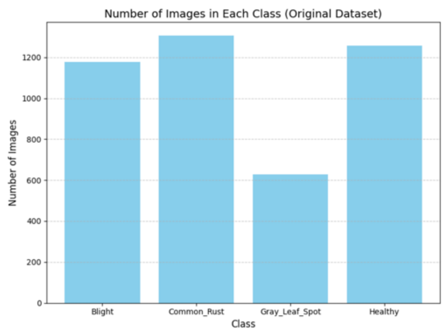

Fig 4 Class Distribution

3.2 The Dataset was then split using Split Folders package.

I divided the data into Train, Validation, and Test sets by 70%, 20%, and 10%, respectively, using the splitfolders package.

```
In [6]: splitfolders.ratio("/content/drive/MyDrive/dataset",output="/content/drive/MyDrive/splitted_data",
                       seed=42,ratio=(.7, .2, .1),group_prefix=None, move=False)
```

Fig 5 Split the dataset.

3.3 Transformation using Data Augmentation

```
# to create training and validation data generators for image data preprocessing.
def train_val_test_data(img_dims, batch_size):

  ## Training data generator with augmentationa
  train_datagen = ImageDataGenerator(rescale = 1./255,
                                     rotation_range = 40, # Randomly rotate images by up to 40 degrees.
                                     width_shift_range = .2,
                                     height_shift_range = .2,
                                     shear_range = .2,# Apply random shearing transformations.
                                     zoom_range = 0.3,
                                     horizontal_flip = True,
                                     vertical_flip=True,
                                     brightness_range=[0.5, 1.5], # Randomly adjust brightness within the specifie
                                     featurewise_center=True,
                                     featurewise_std_normalization=True, # Normalize images based on dataset stand
                                     fill_mode = 'nearest', # Fill missing pixels after transformations with the n
                                 )
```

Fig 6 Data Augmentation code

The ImageDataGenerator module was utilized for data augmentation to take place. Pixel values were rescaled by dividing with 255; adjustments included setting the width and height shift range to 0.2 at the input layer, rotation range of 40 degrees at the rotation layer. An extra 0.2 of shear range, 0.3 zoom range, and x and y axis flipping were further choices meant to help diversify them.

4 Model Training Architecture

Base Model:

- **ResNet-50** & **MobileNetV2-** Load pre-trained weights from ImageNet

- **Custom Layer**: Fully connected layer modified to match the number of classes (4).
- **Key Features for Resnet-50**:
  - Residual blocks for better gradient flow.
  - Batch normalization for stable training.

- **Key Features for MobileNetV2**

  - Exclude the fully connected layers at the top.
  - Ensured the pre-trained layers are not updated during training

Add a fully connected output layer with softmax activation for multi-class classification
Resnet-50 - Parameters used were : Batch Size: 16, Learning Rate: 0.001, Epochs: 25.
The 25 epochs were chosen, as the model started to overfit after 25 epochs. Extended training beyond this would cost more computational time as no growth were seen in the learning curves.

MobileNet-V2 Parameters used were - Batch Size: 128, Learning Rate: 0 .0001,Epochs: 50

Training Loop: The training loop includes monitoring loss and accuracy on both training and validation datasets.

Resnet50 Model creation and training

```
# Define the model
model = models.resnet50(pretrained=True)
model.fc = nn.Linear(model.fc.in_features, NUM_CLASSES)  # Modify output layer
model = model.to(device)

# Loss and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)
```

```
history = {'train_loss': [], 'train_acc': [], 'val_loss': [], 'val_acc': []}
EPOCHS = 25
```

```
# Training Function
def train_model(model, train_loader, val_loader, criterion, optimizer, epochs):

    for epoch in range(epochs):
# Here i am setting the module to the Training mode
        model.train()
        train_loss = 0
        correct = 0
        total = 0
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()
# Here below we will Calculate the number of correct predictions
            _, predicted = outputs.max(1)  # Get class with highest probability
            total += labels.size(0)  # Total number of labels
            correct += predicted.eq(labels).sum().item()   # Count correct predictions

        train_acc = 100. * correct / total
        history['train_loss'].append(train_loss / len(train_loader))
        history['train_acc'].append(train_acc)
```

```
# Below starts the Validation Phase
        model.eval()
        val_loss = 0
        correct = 0
        total = 0
        with torch.no_grad():
            for images, labels in val_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, predicted = outputs.max(1)
                total += labels.size(0)
                correct += predicted.eq(labels).sum().item()

        val_acc = 100. * correct / total
        history['val_loss'].append(val_loss / len(val_loader))
        history['val_acc'].append(val_acc)

        print(f"Epoch [{epoch+1}/{epochs}], Train Acc: {train_acc:.2f}%, Val Acc: {val_acc:.2f}%")
```

Fig 7 Resnet-50 Model Training Code

The Fig 8 depicts the code snippet for searching the parameters to retrain and fine tune the model on best parameters.

```
# Reload the model
NUM_CLASSES = 4
model = models.resnet50(pretrained=True)
model.fc = nn.Linear(model.fc.in_features, NUM_CLASSES)  # Match the number of classes
model.load_state_dict(torch.load("saved_models/resnet50_corn_disease_PRECOPY.pth"))
model = model.to(device)
```

```
learning_rates = [0.0001, 0.0005, 0.001, 0.005]
```

```
optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING_RATE, momentum=0.9)
```

```
batch_sizes = [16, 32, 64]
```

```
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)
```

```
best_val_acc = 0
best_params = {}

for lr in [0.0001, 0.001, 0.01]:
    for batch_size in [16, 32]:
        # Update DataLoader
        train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_workers=4)
        val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False, num_workers=4)

        # Update Optimizer
        optimizer = torch.optim.Adam(model.parameters(), lr=lr)

        # Train Model
        train_model(model, train_loader, val_loader, criterion, optimizer, EPOCHS)

        # Check Validation Accuracy
        val_acc = history['val_acc'][-1]  # Get last validation accuracy

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            best_params = {'learning_rate': lr, 'batch_size': batch_size}
            torch.save(model.state_dict(), "best_tuned_model.pth")

print("Best Parameters:", best_params)
print("Best Validation Accuracy:", best_val_acc)
```

```
Epoch [1/10], Train Loss: 0.2074, Train Acc: 93.62%, Val Loss: 0.0025, Val Acc: 94.74%
Epoch [2/10], Train Loss: 0.1517, Train Acc: 94.23%, Val Loss: 0.0025, Val Acc: 93.18%
Epoch [3/10], Train Loss: 0.1536, Train Acc: 94.85%, Val Loss: 0.0024, Val Acc: 94.98%
Epoch [4/10], Train Loss: 0.1386, Train Acc: 94.98%, Val Loss: 0.0023, Val Acc: 94.38%
Epoch [5/10], Train Loss: 0.1474, Train Acc: 94.85%, Val Loss: 0.0022, Val Acc: 95.69%
Epoch [6/10], Train Loss: 0.1304, Train Acc: 95.09%, Val Loss: 0.0022, Val Acc: 94.74%
```

Fig 8. Searching for the Best parameters

```
EPOCHS = 25
```

```
# Update DataLoader and Optimizer

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True, num_workers=4)
val_loader = DataLoader(val_dataset, batch_size=16, shuffle=False, num_workers=4)
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)

# Train Model with Best Hyperparameters
train_model(model, train_loader, val_loader, criterion, optimizer, EPOCHS)

# Save the final model
torch.save(model.state_dict(), SavedModels_resnet50+"/final_best_modelV3.pth")
print("Model retrained and saved with best hyperparameters.")
```

Fig 9. Hyperparameter Tuning code

Hyperparameter was done using learning rate of 0.0001 and a batch size of 16 because this were found to be the best parameters for our Resnet-50 model. The Model was also saved for later use as google Collab environment gives free access to GPU for limited time.

```python
def mobilenetv2():
# Load the pre-trained MobileNetV2 model
  mobilenet_m = tf.keras.applications.mobilenet_v2.MobileNetV2(input_shape=(
        img_dims,img_dims,3),
        include_top = False, # Exclude the fully connected layers at the top
        weights = 'imagenet' # Load pre-trained weights from ImageNet
      )
  x = mobilenet_m.trainable = False # Ensures the pre-trained layers are not updated during training

  # Get the output of the MobileNetV2 base model
  x = mobilenet_m.output

# Add a global average pooling layer to reduce the spatial dimensions
  x = GlobalAveragePooling2D()(x)

  # Add a fully connected output layer with softmax activation for multi-class classification
  out = Dense(4,activation='softmax')(x)

 # Create the final model by specifying inputs and outputs
  model = Model(inputs = mobilenet_m.inputs, outputs = out)

  # Print the model architecture summary for verification
  model.summary()

  return model
```

```python
from math import ceil # Import the ceil function to round up numbers to the nearest integer

# Calculate the number of steps per epoch for training
steps_per_epoch = ceil(train_gen.samples / batch_size)
validation_steps = ceil(val_gen.samples / batch_size)
# `val_gen.samples`: Total number of validation samples in the dataset
# The same logic as `steps_per_epoch` applies here to cover all validation samples.

# Train the model using the training and validation data generators
history_1 = mobilenetv2_model.fit(
    train_gen,
    steps_per_epoch=steps_per_epoch,  # Number of batches to process per epoch
    epochs=epochs_1,
    validation_data=val_gen,
    validation_steps=validation_steps # Number of validation batches to process per epoch
)
```

```python
# Use the distribution strategy scope for efficient computation on the specified device (GPU/CPU)

with strategy.scope():
  # Create the MobileNetV2 model using the defined function
  mobilenetv2_model = mobilenetv2()
  # Compile the model
  mobilenetv2_model.compile(loss = 'categorical_crossentropy',
                            optimizer = Adam(learning_rate = .0001),
                            metrics = ['accuracy'])
```

Fig 10 MobileNet V2 Model Building Code Snippet

5 Evaluation

```python
# Calculate precision, recall, and F1-score
from sklearn.metrics import accuracy_score
cm = confusion_matrix(true_labels, predicted_labels)

# Example for a single class (class 0)
precision = cm[0, 0] / cm[:, 0].sum()  # TP / (TP + FP)
recall = cm[0, 0] / cm[0, :].sum()    # TP / (TP + FN)
f1_score = 2 * (precision * recall) / (precision + recall)
# Calculate overall accuracy
accuracy = accuracy_score(true_labels, predicted_labels)

print(f"Precision: {precision *100:.2f}% ")
print(f"Recall: {recall*100:.2f}% ")
print(f"F1-Score: {f1_score*100:.2f}% ")
print(f'Accuracy: {accuracy*100:.2f}% ')
```

```
Precision: 94.02%
Recall: 93.22%
F1-Score: 93.62%
Accuracy: 96.36%
```

```python
def plot_learning_curves(history):
    epochs = range(1, len(history['train_loss']) + 1)

    # Plot Training and Validation Loss
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(epochs, history['train_loss'], color='orange', label='Training Loss')
    plt.plot(epochs, history['val_loss'], color='lightblue', label='Validation Loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    # Plot Training and Validation Accuracy
    plt.subplot(1, 2, 2)
    plt.plot(epochs, history['train_acc'], color='orange', label='Training Accuracy')
    plt.plot(epochs, history['val_acc'], color='lightblue', label='Validation Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
    plt.legend()

    plt.tight_layout()
    plt.show()

# Call the function to plot the learning curves
plot_learning_curves(history)
```

Fig 11 Code Snippet for Metrics & Learning Curves

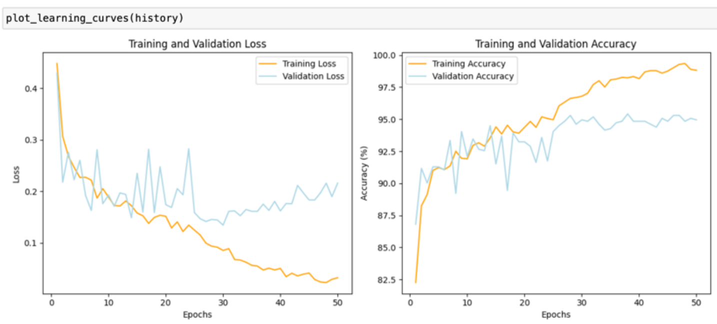

Fig 12 Learning curves of Resnet-50

This code snippet from Figure 11 has been developed to assess the model performance.

Figures 12 illustrate the train and validation evaluation measures of model: accuracy, precision, recall and loss. Only the Evaluation Metrics Graph of Resnet-50 model has been discussed in this setup Manual document.

6 Explainable AI Implementation

The first 2 functions in the Fig 13 code snippet builds a Grad-CAM++ heatmap on a given image based on the target layer (layer4[-1]) for the model while highlighting the regions in the image that is of most importance to the model's prediction. The 3rd function function visualizes the results of the model prediction by showing the original image. As well as the Grad-CAM++ heatmap of where the model's attention is, and a bar chart of the predicted probabilities for each class.

```
#  Perform Model Inference
# This function performs inference for a single preprocessed image.
def perform_inference(model, img_tensor):
    model.eval()
    with torch.no_grad():
        output = model(img_tensor.to(device))
    return output
```

```
# The below function generates a Grad-CAM++ heatmap for a given image
# by focusing on the most relevant regions of the image for the model's predictions,
# using the specified target layer (layer4[-1]) of the model

def generate_gradcam(model, img_tensor):
    cam = GradCAMPlusPlus(model=model, target_layers=[model.layer4[-1]])
                          # , use_cuda=torch.cuda.is_available())
    grayscale_cam = cam(input_tensor=img_tensor)
    return grayscale_cam[0, :]  # Return the first heatmap
```

```
# The below function visualizes the results of a model's prediction by displaying the original image,
# the Grad-CAM++ heatmap for the model's focus, and a bar chart showing the predicted probabilities for each class.

def visualize_result(img, visualization, output, class_names, image_name, figsize=(15, 5)):
    # Convert logits to probabilities
    probabilities = torch.softmax(output.squeeze(), dim=0).cpu().numpy()

    # Predicted class
    predicted_label = probabilities.argmax()

    # Create figure with adjustable size
    fig, axes = plt.subplots(1, 3, figsize=figsize)

    # Plot original image
    axes[0].imshow(img)
    axes[0].axis('off')
#     {class_names[lbl.item()]}
    axes[0].set_title(f"Original: {image_name}", fontsize=18)

    # Plot Grad-CAM++ heatmap
    axes[1].imshow(visualization)
    axes[1].axis('off')
    axes[1].set_title(f"Predicted: {class_names[predicted_label]}", fontsize=18)

    # Plot prediction probabilities
    axes[2].bar(range(len(class_names)), probabilities, color="green")
    axes[2].set_xticks(range(len(class_names)))
    axes[2].set_xticklabels(class_names, fontsize=10)
    axes[2].set_ylim(0, 1)
    axes[2].set_xlabel("Class")
    axes[2].set_ylabel("Probability")
    axes[2].set_title("Prediction Probabilities")

    plt.tight_layout()
    plt.show()
```

Fig 13 Grad CAM ++ Code snippet

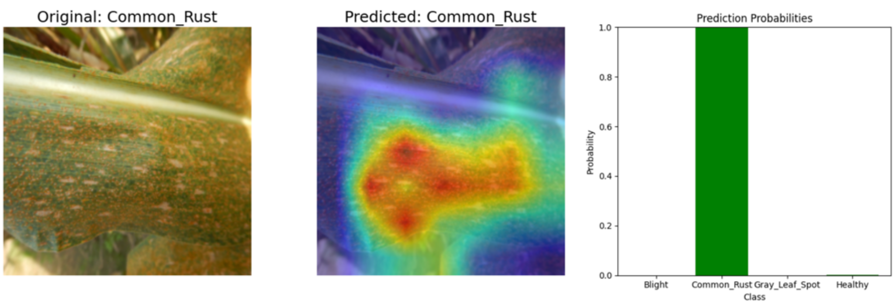

Fig 14 Results for GradCAM ++

```python
def generate_lime_explanations(image_paths, batch_predict_fn, class_names):
    explainer = lime_image.LimeImageExplainer(random_state=42)

    # Create figure with correct size
    num_images = len(image_paths)
    fig, axes = plt.subplots(num_images, 3, figsize=(15, num_images * 5))

    for i, img_path in enumerate(image_paths):
        # Preprocess image
        input_img = preprocess_image(img_path)

        # Generate LIME explanation
        explanation = explainer.explain_instance(
            input_img,
            batch_predict_fn,
            top_labels=len(class_names),
            hide_color=0,
            num_samples=1000
        )

        # Get prediction probabilities and top label
        probs = batch_predict_fn(np.expand_dims(input_img, axis=0))[0]
        top_label = explanation.top_labels[0]

        # Visualize LIME explanation
        img_boundary = visualize_lime_explanation(input_img, explanation, top_label, class_names[top_label])

        # Ensure axes is a 2D array for consistency
        if num_images == 1:
            ax = axes
        else:
            ax = axes[i]

        # Plot Original Image
        ax[0].imshow(input_img / 255.0)
        ax[0].axis('off')
        ax[0].set_title(f"Original: {class_names[top_label]}",fontsize=18)

        # Plot LIME Highlighted Image
        ax[1].imshow(img_boundary)
        ax[1].axis('off')
        ax[1].set_title(f"Predicted LIME: {class_names[top_label]}",fontsize=18)

        # Plot Prediction Graph for All Classes
        ax[2].bar(range(len(class_names)), probs, color='green')
        ax[2].set_xticks(range(len(class_names)))
        ax[2].set_xticklabels(class_names)
        ax[2].set_ylim(0, 1)
        ax[2].set_xlabel("Classes")
        ax[2].set_ylabel("Probability")
        ax[2].set_title("Prediction Probabilities",fontsize=18)

    plt.tight_layout()
    plt.show()
```

Fig 15 Lime Code snippet

This function computes and shows LIME (Local Interpretable Model-agnostic Explanations) explanations for a set of input images including the original image, the image areas that affect the prediction according to LIME and prediction probabilities for each class.

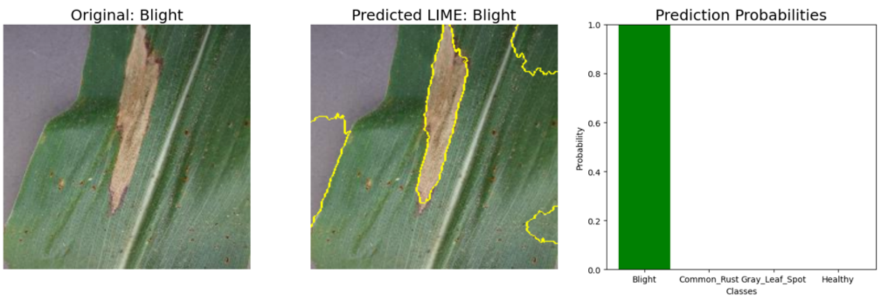

Fig16 Result for LIME Explanations