# 1004 - JUTE PEST


## Team Nash

- Amirtesh Raghuram
- Nigam Parida
- Shreya Pattanayak
- Harish Sathyanandan


*Transforming the Future*


## Guide’s Name

Nikhil Maurya

# Goal

**1. Enhanced Crop Management:** By identifying and classifying different pests, farmers can implement timely and targeted pest control measures, enhancing overall crop yield and quality.

**2. Sustainable Agriculture:** Developing accurate pest detection models promotes sustainable farming practices by minimizing the use of harmful pesticides, promoting ecological balance, and reducing environmental impact.

**3. Economic Benefits:** Early detection of pest infestations can save farmers from potential losses and reduce the costs associated with crop damage and pest management.

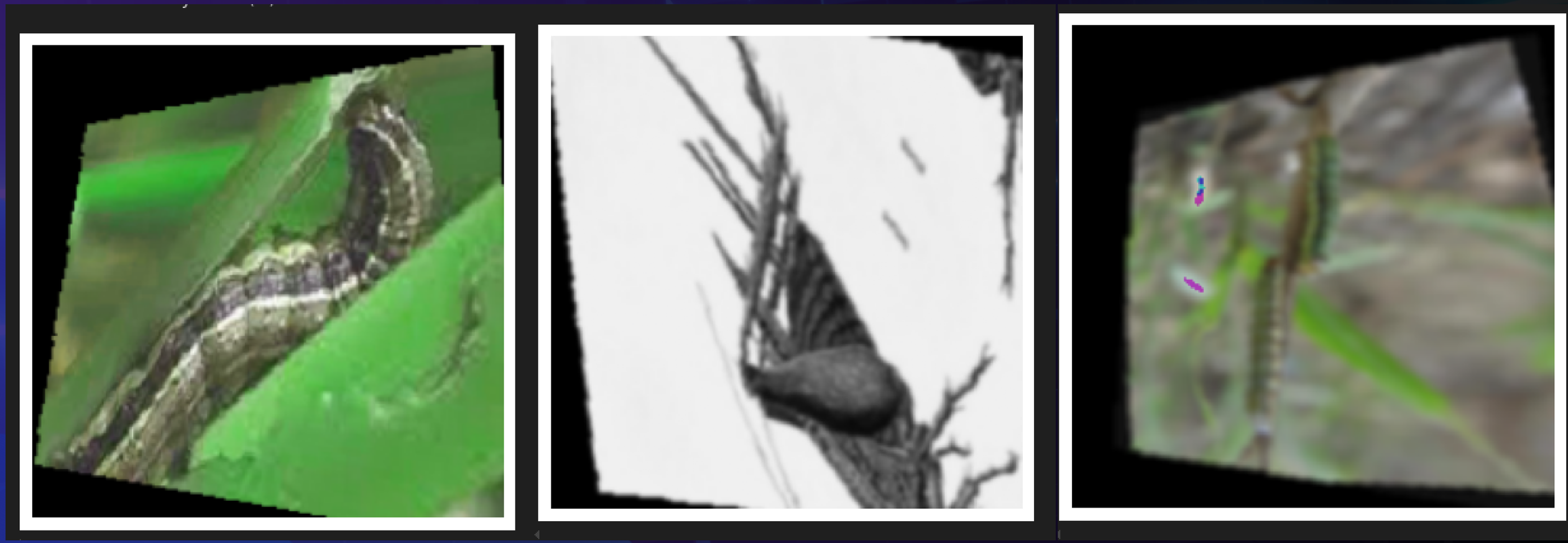
**4. Informed Decision-Making:** The insights gained from analyzing pest trends can help agricultural stakeholders make data-driven decisions regarding crop rotation, pest-resistant plant varieties, and resource allocation.

**5. Contribution to Research:** This project can contribute to the broader field of agricultural research and technology, providing foundational data for further studies on pest behavior, resistance, and agricultural innovation.

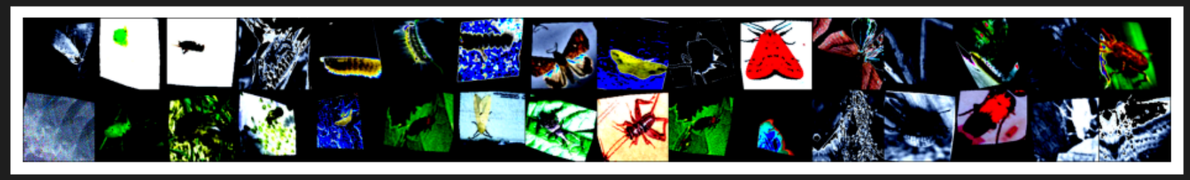
# Preprocessing of Images

The preprocessing we have followed includes applying a series of transformations to images to prepare them for model training, testing, and validation. Training images undergo extensive augmentation, including resizing, cropping, horizontal and vertical flipping, random rotations, color adjustments, perspective distortion, blurring, grayscale conversion, and other variations to introduce diversity and prevent overfitting. These images are then converted to tensors and normalized based on standard mean and standard deviation values. For validation and test images, simpler transformations like resizing, cropping, and normalization are applied, ensuring consistency during evaluation without excessive alterations.

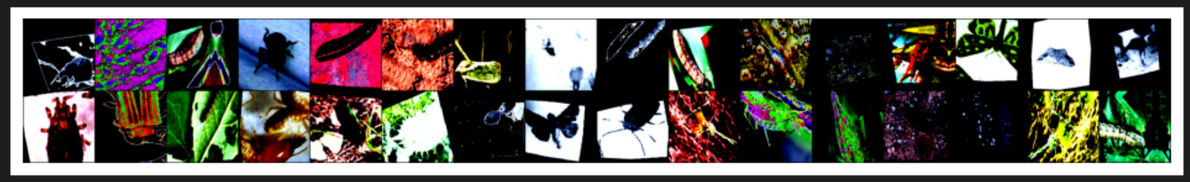
# Looking at Few of the Images Used in Training



```
eda.show_batch(train_dl)
```



```
eda.show_batch(train_dl)
```



Noticing these images, we can conclude that the process of augmenting the training dataset has been successful as the normal RGB images which are present in the dataset has many variation now such as randomly being flipped and jittered among many other transformations. This is an essential process since augmentation of images is a good way to increase the variability in our datasets, which can help improve model performance.

# Training of Model

To train the model efficiently, we have used to pre trained efficientnet_b0 model which is present in torchvision by default. EfficientNet-B0 is popular in PyTorch because it strikes a good balance between accuracy and speed. It’s designed to scale its depth, width, and resolution in a smart way, so it performs well without being too heavy on resources. This makes it a great choice for getting strong results with less computational power.

```python
class Model(ImageClassificationModel):
    def __init__(self, out_size):
        super().__init__()

        self.efficientnet = models.efficientnet_b0(weights='DEFAULT')

        num_features = self.efficientnet.classifier[-1].in_features

        self.efficientnet.classifier[-1] = nn.Linear(num_features, out_size)

    def forward(self, x):
        x = self.efficientnet(x)
        return x
```

The AdamW optimizer, present within torch.optim module of PyTorch was chosen with the learning rate as 0.001 and weight decay of 1e-4. AdamW is often used in PyTorch because it’s an improved version of the Adam optimizer. By separating weight decay from the gradient update, it helps prevent the model’s weights from becoming too large, leading to better performance. It keeps the benefits of Adam, like adaptive learning rates, while making regularization more effective, especially for bigger models.

Loss function chosen was CrossEntropyLoss, which is present withing the torch.nn module of PyTorch.

In PyTorch, a learning rate scheduler helps control how fast a model learns during training. It adjusts the pace as training progresses, making sure the model learns efficiently and doesn’t get stuck making the same mistakes. The one-cycle learning rate (torch.optim.lr_scheduler.OneCycleLR) is used because it makes training a model faster and more effective. It starts with a high learning rate, which helps the model quickly explore and find better solutions. Then, as training goes on, it gradually lowers the learning rate, allowing the model to fine-tune its understanding and improve accuracy. The parameters used in the learning scheduler were the optimizer passed in, max_lr of learning rate used in optimizer, epochs of 20 (same as number of epochs used in training model) and steps_per_epoch as the length of train dataset loader.

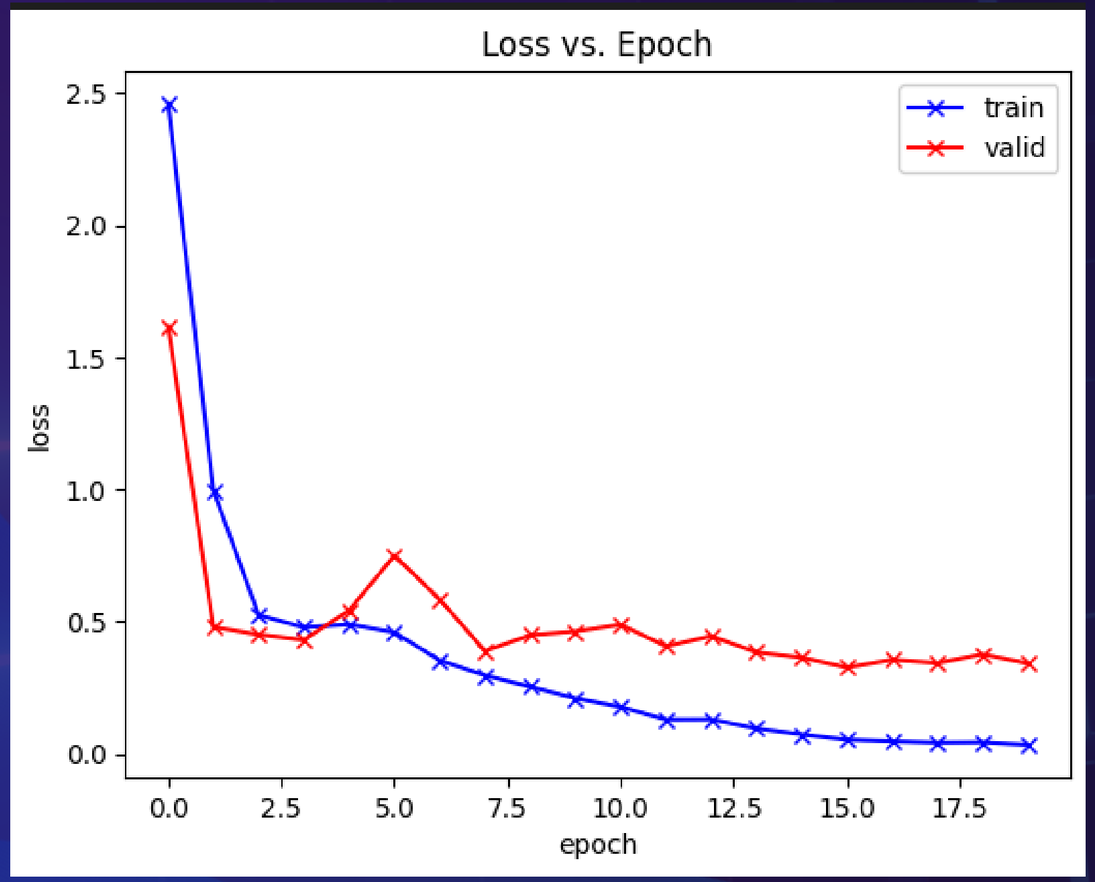
# Training Progress

```python
import time
start=time.time()

history=model.fit(epochs=epochs,max_lr=max_lr,train_loader=train_dl,val_loader=valid_dl,
                  weight_decay=weight_decay,grad_clip=0.1,
                  opt_func=optim.AdamW)

print(f'Training time: {(time.time()-start)/60} minutes')
```

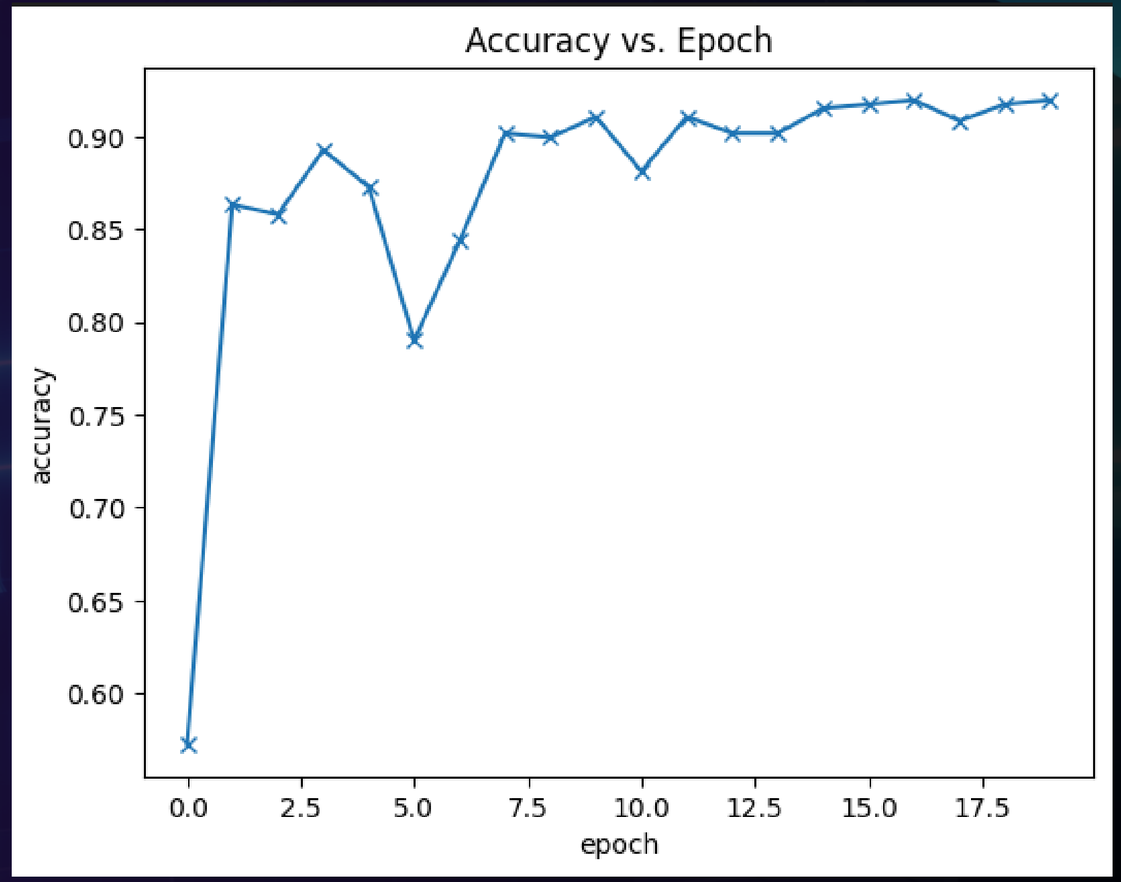
```
Epoch [0], train_loss: 2.4590, train_acc: 0.2925, val_loss: 1.6161, val_acc: 0.5724
Epoch [1], train_loss: 0.9964, train_acc: 0.7247, val_loss: 0.4799, val_acc: 0.8634
Epoch [2], train_loss: 0.5245, train_acc: 0.8404, val_loss: 0.4508, val_acc: 0.8580
Epoch [3], train_loss: 0.4784, train_acc: 0.8510, val_loss: 0.4311, val_acc: 0.8929
Epoch [4], train_loss: 0.4900, train_acc: 0.8479, val_loss: 0.5424, val_acc: 0.8728
Epoch [5], train_loss: 0.4594, train_acc: 0.8574, val_loss: 0.7505, val_acc: 0.7902
Epoch [6], train_loss: 0.3538, train_acc: 0.8876, val_loss: 0.5843, val_acc: 0.8438
Epoch [7], train_loss: 0.2970, train_acc: 0.9078, val_loss: 0.3890, val_acc: 0.9018
Epoch [8], train_loss: 0.2541, train_acc: 0.9186, val_loss: 0.4487, val_acc: 0.8996
Epoch [9], train_loss: 0.2102, train_acc: 0.9323, val_loss: 0.4630, val_acc: 0.9107
Epoch [10], train_loss: 0.1774, train_acc: 0.9425, val_loss: 0.4887, val_acc: 0.8812
Epoch [11], train_loss: 0.1295, train_acc: 0.9579, val_loss: 0.4069, val_acc: 0.9107
Epoch [12], train_loss: 0.1300, train_acc: 0.9599, val_loss: 0.4452, val_acc: 0.9018
Epoch [13], train_loss: 0.0961, train_acc: 0.9690, val_loss: 0.3843, val_acc: 0.9018
Epoch [14], train_loss: 0.0732, train_acc: 0.9750, val_loss: 0.3642, val_acc: 0.9152
Epoch [15], train_loss: 0.0536, train_acc: 0.9819, val_loss: 0.3269, val_acc: 0.9174
Epoch [16], train_loss: 0.0475, train_acc: 0.9844, val_loss: 0.3558, val_acc: 0.9196
Epoch [17], train_loss: 0.0424, train_acc: 0.9849, val_loss: 0.3437, val_acc: 0.9085
Epoch [18], train_loss: 0.0435, train_acc: 0.9869, val_loss: 0.3764, val_acc: 0.9174
Epoch [19], train_loss: 0.0343, train_acc: 0.9872, val_loss: 0.3423, val_acc: 0.9196
Training time: 17.212519284089407 minutes
```

The image shows how the model is trained over 20 epochs, with both training and validation metrics displayed. At first, the training loss is quite high, but it gradually drops as the model learns from the data, which is a good sign. The validation accuracy also improves over time, showing that the model is getting better at making predictions on new data it hasn't seen before. This balance between training and validation is important because it means the model isn’t just memorizing but actually learning. Overall, the training took around 17.2 minutes, indicating a smooth and effective learning process.

# Evaluating Model Metrics After Training Process




Train and validation loss over the course of training




Accuracy on validation dataset over the course of training

# Model Performance on the Test Data

## (Completely Unseen Data)

```
accuracy,f1,precision,recall=eval.evaluate(test_dl)
print(f'Test dataset metrics: ')
print(f'Accuracy: {accuracy*100}%')
print(f'F1 Score: {f1*100}%')
print(f'Precision Score: {precision*100}%')
print(f'Recall Score: {recall*100}%')
```

```
Test dataset metrics:
Accuracy: 98.68073878627969%
F1 Score: 98.65884848106793%
Precision Score: 98.66946778711485%
Recall Score: 98.72549019607843%
```

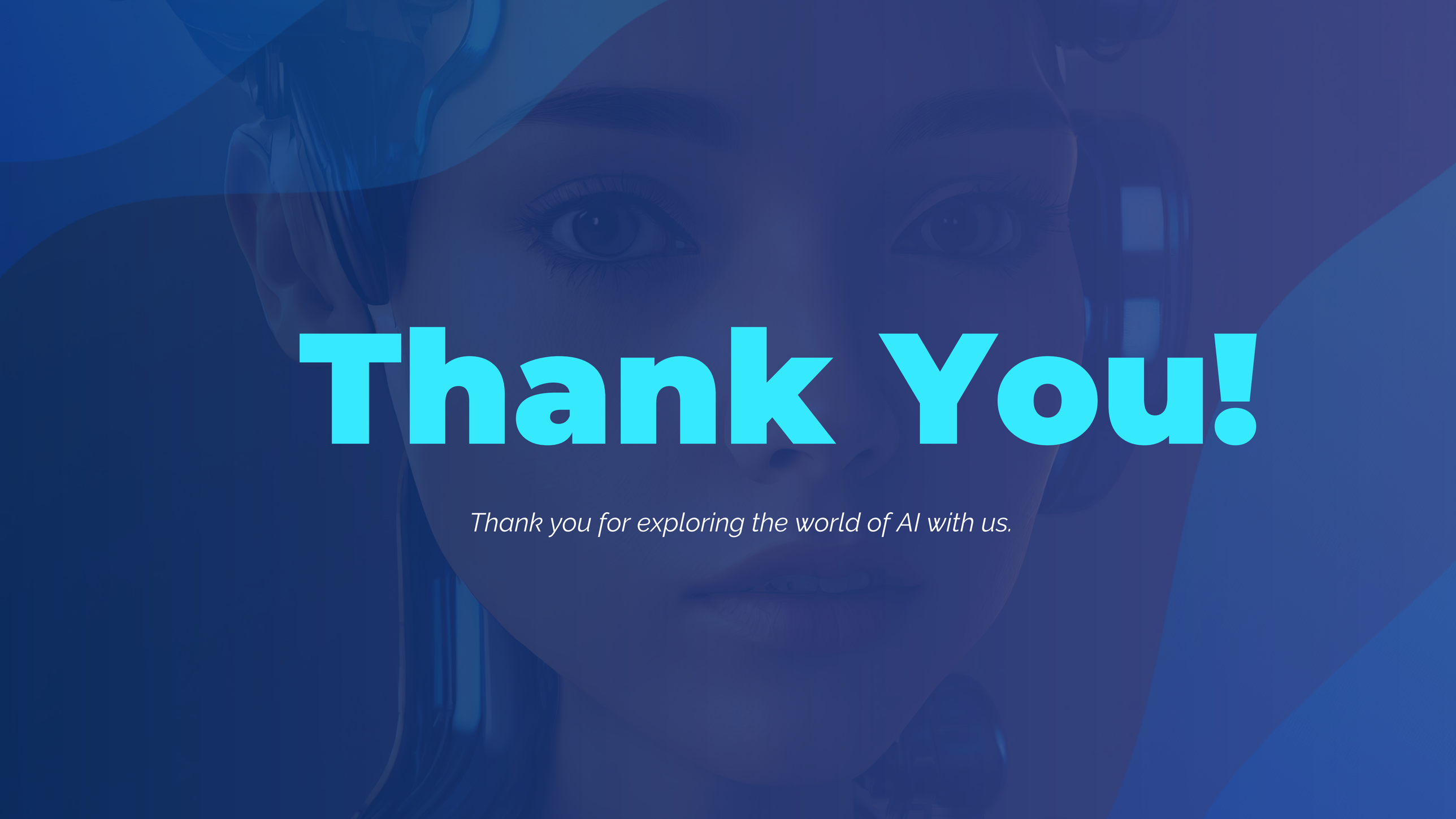
The image shows the evaluation metrics for a machine learning model that has been tested on a dataset. With an accuracy of about 98.68%, it's clear the model is doing a fantastic job of correctly classifying almost all test samples. The F1 score, which gives a balanced view of precision and recall, is around 98.65%. This means the model is good at avoiding both false positives and false negatives.

Moreover, the precision and recall scores are also impressive, both above 98%. This tells us that not only is the model making accurate predictions, but it's also consistent and reliable in its performance.

# Making a Few Predictions with the Model on Random Images from the Test Data

| | Index | Actual | Predicted |
|---|---|---|---|
| 281 | 281 | Pod Borer | Pod Borer |
| 170 | 170 | Jute Stem Girdler | Jute Stem Girdler |
| 209 | 209 | Jute Stem Weevil | Jute Stem Weevil |
| 324 | 324 | Termite | Termite |
| 163 | 163 | Jute Semilooper | Jute Semilooper |
| 40 | 40 | Cutworm | Cutworm |
| 110 | 110 | Jute Hairy | Jute Hairy |
| 139 | 139 | Jute Red Mite | Jute Red Mite |
| 42 | 42 | Cutworm | Cutworm |
| 95 | 95 | Jute Aphid | Jute Aphid |

As we can see, the model is able the predict the actual pest excellently on about 10 images from the unseen test dataset, indicating that the model is performing really well.



# Thank You!

*Thank you for exploring the world of AI with us.*