Northeastern University
Khoury College of Computer Sciences

BugBot: Using Deep Learning For Household Pest Image Classification

Author(s): Shirley Fong, Keegan Veazey, Le Ju

Northeastern University

DS 5500 Capstone: Application in Data Science

Spring 2025

Date: April 16, 2025

Abstract

Insects are a common nuisance, and many species appear similar which makes identification difficult without expert assistance. Many homeowners in the United States rely on professional pest control services, which can cost thousands of dollars. Rising costs have led homeowners to attempt pest-identification themselves. To address this issue, this research focuses on developing an image classification model to identify common New England household pests using transfer learning with convolutional neural networks (CNNs). Data was collected through programmatic web scraping and supplemental sources such as Google, Reddit, and iNaturalist. The final dataset contains 11 pest classes with approximately 160 photos per class. The raw data was then preprocessed using manual filtering, image hashing, resizing, recoloring, and padding. The dataset was then divided into training, validation, and test sets. Data augmentation was then applied to the training set to increase the data set size without data leakage. For classification, a CNN was trained on 11 pest classes. We utilize three pre-trained models for transfer learning: DenseNet201, MobileNetV2, and Xception. Batch normalization, dropout layers, Bayesian optimization and random search hyperparameter optimization is also employed. Performance is assessed using accuracy, precision, recall, F1 score, confusion matrices, loss and accuracy curves, AUC-ROC curves, and one-versus-rest (OVR) scores. The workflow is illustrated in *Figure 2.2: BugBot Workflow* and details the interactions between the various components of this research.

Contents

Chapter 1

Introduction

1.1 Problem Specification

Household pests are often visually similar, making it difficult for non-experts to correctly identify them. This visual ambiguity frequently leads to misidentification, which in turn can result in ineffective treatment methods, unnecessary costs and time waste, and/or health risks. Currently, there is no widely accessible, accurate, image-based tool specifically designed for identifying New England common household pests, especially using smartphone images.

Our project, BugBot, addresses this gap by developing a deep learning model capable of classifying 11 common New England household pest species (such as but not limited to rice weevil, brown stink bug, and subterranean termite) using convolutional neural networks with transfer learning. The model is trained on data that was web-scraped from a variety of sources (e.g., Reddit, iNaturalist) to better reflect user-uploaded smartphone images.

In developing BugBot, several challenges were addressed, including a small dataset size, significant visual similarity among certain pest species, and variations in image lighting and background noise. The need to balance model accuracy with computational efficiency, in addition to creating a Web application, was also a main challenge.

1.2 Objectives

The primary objective of this project is to build a reliable and accessible image classification tool that assists everyday people in identifying household pests quickly and accurately. This allows property owners, mangers, renters, and others to detect and treat pests early, potentially reducing health hazards and pest control costs.

Specifically, we aim to:

- Implement and compare three transfer learning CNN architectures (DenseNet201, MobileNetV2, Xception) for pest classification.
- Enhance model performance through rigorous data preprocessing, augmentation, and hyperparameter optimization (using Random Search and Bayesian Optimization).
- Evaluate performance using comprehensive metrics (accuracy, precision, recall, F1, AUC-ROC, and one-vs-rest scores) and reduce overfitting through regularization and dropout strategies.

Ultimately, the project aims to validate whether a simple yet robust CNN model can deliver high-performance pest classification in real-world home environments.

1.3 Scope

The scope of this project is centered on the classification of 11 specific household pest species commonly found in the New England region. The focus is exclusively on pests in indoor or home-adjacent settings, rather than agricultural contexts. The dataset comprises real-world user-uploaded images with variability in lighting, angles, and resolution, reflecting the conditions of smartphone photography by non-expert users.

The project is designed to function on standard computing environments, with practical deployment via a user-friendly web interface (Streamlit). This environments allows users to upload pest images and receive real-time classification feedback. While the current scope is limited to 11 pest types, the methodology and pipeline are scalable to additional species and regions in future iterations.

1.4 Related Works

Recent studies have demonstrated the effectiveness of CNNs in pest classification across various contexts. A relevant study is the research paper “An Efficient Deep Learning Approach for Jute Pest Classification Using Transfer Learning” by Muhammad Tanvirul Islam and Md. Sadekur Rahman. The primary similarity between their work and our proposed approach lies in the pest-classification specific image topic as well as the use of a CNN model use. However, their research specifically targets Jute pests, which pose significant challenges to Jute crop production in Bangladesh (Islam & Rahman, 2024). In contrast, our study will use a completely different dataset and will be used to classify common household pests found in New England.

In our review, we found the lack of diversity in training data to be notable factors across the literature. For example, Turkoglu et al.'s PlantDiseaseNet study in the paper "PlantDiseaseNet: Convolutional Neural Network Ensemble for Plant Disease and Pest Detection" utilized a highly controlled dataset, comprised of RGB images captured with standardized equipment and consistent resolution of 4000 x 6000 pixels. This contrasts significantly with our proposed methodology, which will incorporate web-scraped images with varying resolutions and background contexts. Furthermore, PlantDiseaseNet uses an ensemble of Support Vector Machines (SVMs) as the primary classifying component, only using a CNN as an averaging model for the final classification across multiple SVM outputs (Turkoglu et al., 2021). Meanwhile, our approach uses the CNN directly by feeding in the image data as the first layer.

The literature also reveals a consistent emphasis on data processing to improve model performance. For instance, Rehman et al.'s work on brain tumor classification in the paper "A Deep Learning-Based Framework for Automatic Brain Tumors Classification Using Transfer Learning", while in a different domain, provides valuable insight into image pre-processing. Their application of image flipping and rotation techniques (Rehman et al., 2019) aligns with our planned approach, though our implementation must account for the greater variability in pest imagery compared to standardized medical imaging.

The research paper "Crop pest classification based on deep convolutional neural network and transfer learning" by the authors K. Thenmozhi and U. Srinivasulu Reddy also implements image processing for improved model performance, applying reflection, scaling, and rotation techniques. Thenmozhi and Reddy's research also represents one of the more comprehensive studies in the field, achieving over 95% classification accuracy across three different datasets of crop pests (Thenmozhi & Srinivasulu Reddy, 2019). They further compare custom CNN models against pre-trained architectures including AlexNet and ResNet. We intend to adapt their comparison approach and use similar augmentation techniques while acknowledging the different challenges presented by our more diverse dataset.

For contrast, Rajan et al.'s work in the research paper "Detection And Classification Of Pests From Crop Images Using Support Vector Machine" demonstrates the limitations of simpler classification approaches. Their SVM-based system, while effective for their use case of binary classification between whiteflies and aphids, highlights the need for more complex learning approaches when handling multiple pest classes. Rajan et al.'s approach relies on manually designed feature extraction to detect and recognize pests. Manual feature extraction limits the accuracy and adaptability of pest detection by failing to capture varied image characteristics (Guo et al., 2024). Furthermore, their study's reliance on controlled greenhouse imagery further underscores the importance of our more comprehensive approach using diverse image sources.

Our research aims to bridge the gap between these agricultural-focused studies and household pest identification needs by incorporating the successful methodological elements identified in the literature such as image rotation and mirroring, and by comparing our CNN model to similar projects such as Islam and Rahman (2024) jute pest study. Our planned use of web-scraped images and utilizing multiple pre-trained models represents a novel approach that addresses everyday people's needs while building upon established technical foundations.

Chapter 2

Technical Approaches

2.1 Experimental Setup

2.1.1 Hardware and Software

The experiments were conducted on a high-performance laptop powered by an Intel Core i9-12900H processor from Intel's 12th Generation Alder Lake series. This processor features 14 cores, including 6 performance cores (P-cores) and 8 efficiency cores (E-cores), with a total of 20 threads enabled via Hyper-Threading Technology. The base clock speed of 2.50 GHz can turbo boost up to 5.0 GHz, providing high computational power for deep learning tasks. The system is equipped with 32GB of DDR4 RAM, ensuring smooth execution of computationally intensive tasks such as training neural networks and hyperparameter tuning.

For model training, we used a NVIDIA GeForce RTX 3080 Ti GPU with 12GB of GDDR6 VRAM. This GPU offers high-speed parallel computing with dedicated CUDA cores and Tensor cores optimized for deep learning acceleration. The experiments were conducted on a 1TB NVMe SSD, ensuring fast data access and reducing storage bottlenecks.

The software stack included TensorFlow-GPU 2.x, configured to leverage CUDA and cuDNN for GPU acceleration. CUDA 11.x was installed to enable efficient parallel computing, while cuDNN 8.x was used to accelerate convolutional operations and deep learning computations. Hyperparameter tuning was performed using Keras Tuner, utilizing Bayesian Optimization and Random Search to find optimal model parameters. Data preprocessing and augmentation were implemented using TensorFlow's tf.data API, while TensorFlow's ImageDataGenerator was used for image augmentation. Model evaluation and visualization were done using Matplotlib, Sklearn, and Tensorflow to track training metrics and assess performance.

2.2 Model Selection

To evaluate model effectiveness, we initially ran seven different models: CNN without transfer learning, EfficientNetB0, ResNet50, VGG16, MobileNetV2, DenseNet201, and Xception. The models were selected based on Islam et al.'s jute pest agricultural study "An Efficient Deep Learning Approach for Jute Pest Classification Using Transfer Learning" (Islam & Rahman, 2024) and Talukder et al.'s jute pest paper "JutePestDetect: An Intelligent Approach for Jute Pest Identification Using Fine-Tuned Transfer Learning" (Talukder et al., 2023). These papers inspired our idea for creating a model that could detect common household pests.

Based on the validation metrics, we filtered the original seven models to the following three CNN architectures: DenseNet201, MobileNetV2, and Xception. DenseNet201. These models had the best performances regarding evaluation metrics. DenseNet201 has the ability to strengthen feature propagation and reduce redundant parameters, and it makes it highly effective for classification tasks with limited data. MobileNetV2 had the second-best performance and was selected for its architecture, which balances accuracy and efficiency, making it ideal for mobile applications. Lastly, Xception had the third-best performance in accuracy.

2.3 Dataset and Preprocessing

The majority of the BugBot dataset was scraped using a publicly available API for scraping images from Bing search queries. A manual review and filtering process was then applied to the Bing-based dataset and was subject to the following standards:

1. The insect must be present in the image
2. The image must show at least 75% of the insect's body
3. The photo is not a drawing, cartoon, or an AI generated image
4. The image depicts the adult/matured insect

Duplicate Elimination and Data Split

For all collected images, it is also essential to eliminate duplicates or near-duplicates to prevent data leakage. Visual inspection in addition to image hashing are implemented. Since no temporal or spatial dependencies exist in the dataset, random splitting for training, validation, and test subsets is appropriate. The validation set will guide hyper-parameter tuning, while the test set will provide an unbiased assessment of the model's performance.

Group and Colony Pests

Furthermore, it is important to note the unique nature of many pests that usually infest home environments in groups or colonies, including ants, termites, and bed bugs. Therefore, a direct effort was made to collect additional group images of insect infestations to be included in the dataset. After filtering, a range of approximately 16%-76%, depending on the insect class, was kept from the original scraped data based on the above criteria.

Manual Search Data Supplement

After web scraping, additional data was collected by performing manual Google searches for image results using insect keywords, and by taking advantage of community postings on websites such as Reddit to collect home-environment-specific images. After combining the scraped images and manual supplement, the resulting dataset includes 160 raw images across each of the 11 insect classes.

Data Features

Since our dataset consists of RGB images with x and y coordinates, the only features are the image pixels themselves. However, additional hidden features will be extracted through preprocessing and modeling the data and includes visual characteristics, such as texture, RBG value, and pixel sequence. Furthermore, when the complete image matrix is flattened into a 1-dimensional feature vector, each pixel element becomes a feature of the overall image.

Target Feature and Balanced Classes

The target variable in this supervised learning project is the insect class, representing one of the 11 common household pests. It is a well-defined categorical variable with clear labels derived from the dataset. Since the dataset is designed with a balanced distribution of 160 images per class (100 for training, 40 for validation, and 20 for testing), there is no inherent class imbalance, and resampling techniques are not necessary. This ensures that the model will not favor any single class during training or evaluation.

Due to the dataset design and manual filtering criteria, we have curated a dataset that adequately represents every insect class to ensure equal distribution of training, validation, and test data to reduce bias. We are also aware of potential bias in web scraping data due to the prevalence of scientific images which may not reflect the targeted end user – an everyday homeowner classifying an insect in their home. To mitigate this we have further supplemented the dataset with manually selected images with diverse backgrounds and contexts including images from Reddit and iNaturalist.

Resources Required

In terms of resources required, the dataset, consisting of 1,760 images, is relatively small and is manageable to process with standard resources. The dataset size even after augmentation does not exceed the limits of local storage or memory. Therefore, the dataset does not require advanced techniques such as distributed processing (e.g., using Apache Spark or Dask) since the computational and storage demands are minimal.

Data Preprocessing Steps

Although no distributed processing will be required, the data requires multiple pre-processing steps which are summarized by *Figure 2.1: Data augmentation techniques*, and are described below:

1. Image hashing to remove duplicates
2. Standardizing the data to a fixed image size (e.g., 224x224)
3. Normalizing the pixel values
4. Color standardization (i.e., RGBA $\rightarrow$ RGB)
5. Adding padding to the images
6. Applying data augmentation techniques including rotation, height and width shift, and brightness adjustments to expand the training dataset

Addressing Potential Limitations

It is important to acknowledge the limitations of this dataset which could include a lack of sufficient data, potentially leading to over-fitting during training and data imbalance. When this case occurs, it could skew model predictions and affect the model's accuracy. This will be addressed and mitigated through data augmentation, mentioned above, through techniques including image rotation, horizontal and vertical flips, and cropping of images. By ensuring that each insect class contains 160 unique images and by expanding the dataset to include varied versions of all images, we effectively diversify the data set to mitigate over-fitting and enhance the model performance.

2.4 Training and Validation Process

This section discusses the architecture, training process, and hyperparameter-tuning methodology for BugBot. As discussed in the previous section (*Section 2.2: Model Selection*), the three aforementioned pre-trained models (DenseNet201, MobileNetV2, and Xception) were employed to accomplish the transfer learning portion of our modeling. By using these models for transfer learning, we leverage

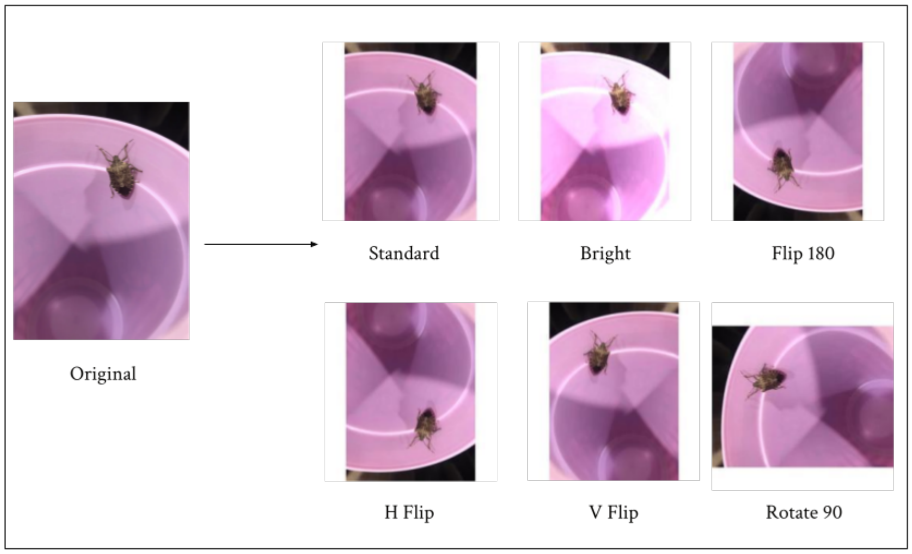


Figure 2.1: Data augmentation techniques

the generalized knowledge pre-trained on ImageNet's diverse 1.4 million images for each model. This approach reduces computational resources and training time and is accomplished by freezing the base model's first four dense blocks and training our custom classification head. By employing transfer learning, we are able to focus further resources on hyperparameter optimization rather than designing complex networks from scratch.

2.4.1 Model Architecture and Configuration

Transfer Learning Base Model

To configure each transfer learning model with a custom classification head, the following structure was used:

For each baseline model (DenseNet201, MobileNetV2, Xception):

1. Use preprocessed training data as input (for more information about this step, see Figure 2.2: BugBot Workflow, Section 2.3: Dataset and Preprocessing, and Section 2.4.2: Data Preprocessing and Data Split)
2. Set baseline model as an initial base, freezing ImageNet weights
3. Append customized classification head to the base model

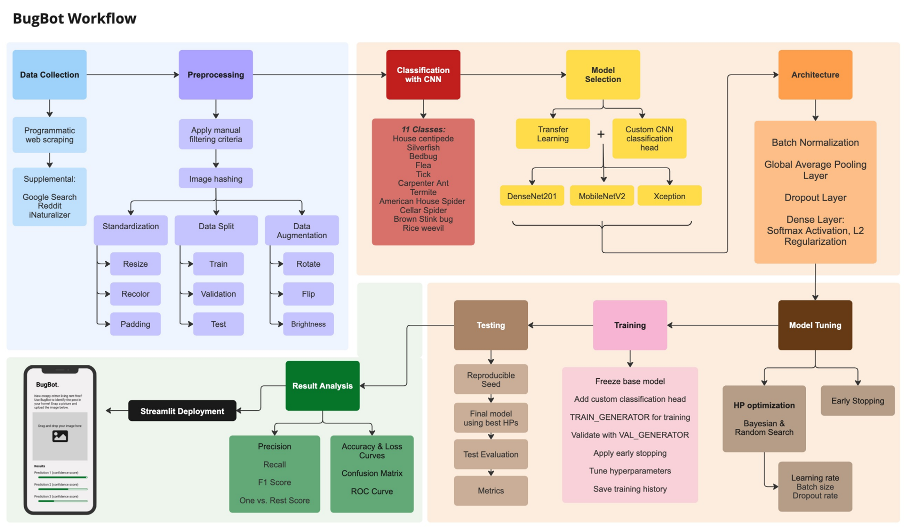


Figure 2.2: BugBot Workflow

Classification Head

In designing the final custom classification head, layer, and regularization experiments were conducted to find the level of complexity needed to balance metrics of interest (see Section 4.1: Evaluation Metrics) and overfitting (see Section 2.4.3: mitigating Overfitting for more detail). The final classification head consists of the following layers:

- Batch normalization layer to help with the domain shift issue and overfitting
- Global Average Pooling layer to reduce spatial dimensions while preserving channel information
- Dropout layer with tunable rate for regularization
- Dense output layer with softmax activation, output neuron count matching the number of pest classes (11 classes)

Final Model Architecture

Figure 2.3: Final Model Architecture visualized BugBot's final model. As discussed in Chapter 4 and specifically the results discussion in sections 4.2 - 4.3, the tuned DenseNet201 model performed best of the three aforementioned models in all quantitative and qualitative metrics. The optimized hyperparameters were found using hyperparameter tuning, which is discussed in Section 2.4.5: Hyperparameter Tuning and Optimization Algorithms.

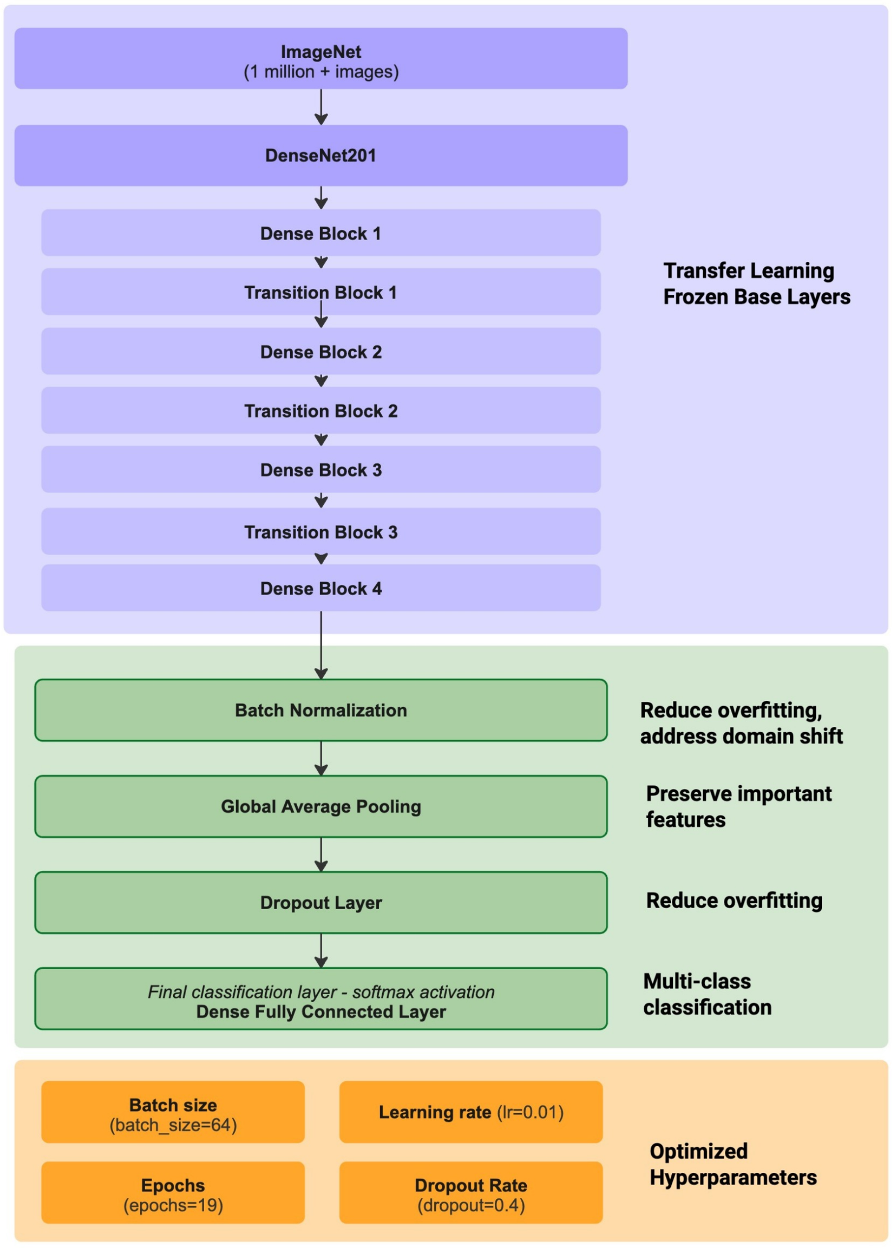


Figure 2.3: Final Model Architecture

2.4.2 Data Preprocessing and Data Split

Before training, the data is preprocessed using the preprocessing steps described in Section 2.3: Dataset and Preprocessing and importantly includes standardization measures of resizing, scaling, recoloring, and removing duplicates via image hashing with the Undouble library (see Section 3: Data Collection and Preparation).

Training, Validation, Test Split

The data is then divided into three separate directories: training, validation, and testing sets, following the split of 100 training images, 40 validation images, and 20 test images for each class. The training set then undergoes data augmentation, including image mirroring, brightness adjustment, and rotation using PIL Image, ImageEnhance, and Numpy libraries. Notably, for each dataset directory, Keras ImageDataGenerator library is used to create custom data generators for efficiency throughout the model training and evaluation process:

- `TRAIN_GENERATOR`: Provides shuffled batches for model training
- `VAL_GENERATOR`: Provides validation data during training and hyperparameter tuning
- `TEST_GENERATOR`: Provides unshuffled data for final model evaluation
- `EVAL_VAL_GENERATOR`: Provides unshuffled validation data for metric calculation

Preprocessing Pipeline

For reproducibility and functionality, an automated data preprocessing pipeline script was created to run the full set of preprocessing steps in a single script call. This pipeline is visualized by the upper left blue section of Figure 2.2: BugBot Workflow and contains functions denoted by the blue and purple ”Data Collection” and ”Preprocessing” trees. Note that the pipeline does not perform the manual filtering described in Section 2.3: Dataset and Preprocessing, since this step has already been completed by the BugBot team. The pipeline can be run following the instructions provided in the BugBot README.md in the BugBot GitHub repository (see Section 6.1: GitHub).

2.4.3 Mitigating Overfitting

AS previously mentioned, when training a CNN on a small dataset, the risk of overfitting can be high since the model can memorize the training data instead of generalizing to unseen data. Therefore, mitigating overfitting and performing hyperparameter tuning for better model performance and efficiency was essential in our model experiments to ultimately arrive at the final transfer learning (base DenseNet201) model (see *Figure 2.3: Final Model Architecture*). To do this, the models are configured with batch normalization layers, dropout layers, and employ early stopping.

Batch Normalization and Dropout Layers

Batch normalization helps by keeping the activations of each layer stable during training. Dropout helps by randomly dropping neurons during training, preventing reliance on specific features (importantly, note the random seeds for the libraries: random, numpy, tensorflow, and keras are defined for reproducibility). Dropout rate is defined as a hyperparameter and is tuned using Keras Tuner (see Section 2.4.4: Hyperparameter Tuning and Optimization Algorithms).

Early Stopping

Early stopping, unlike the other methods mentioned, is not a layer application and instead contributes to mitigating overfitting by preemptively stopping the training process when the metric of choice, we use validation loss, stops improving.

The early stopping parameters we implemented specify the following:

- Metric: Validation loss
- Patience: Number of epochs with no improvement allowed before stopping training. Set *patience = 3*.
- Minimum Delta Value: Minimum change in the monitored metric required to consider it an improvement. Set *min_delta = 0.001*

2.4.4 Hyperparameter Tuning and Optimization Algorithms

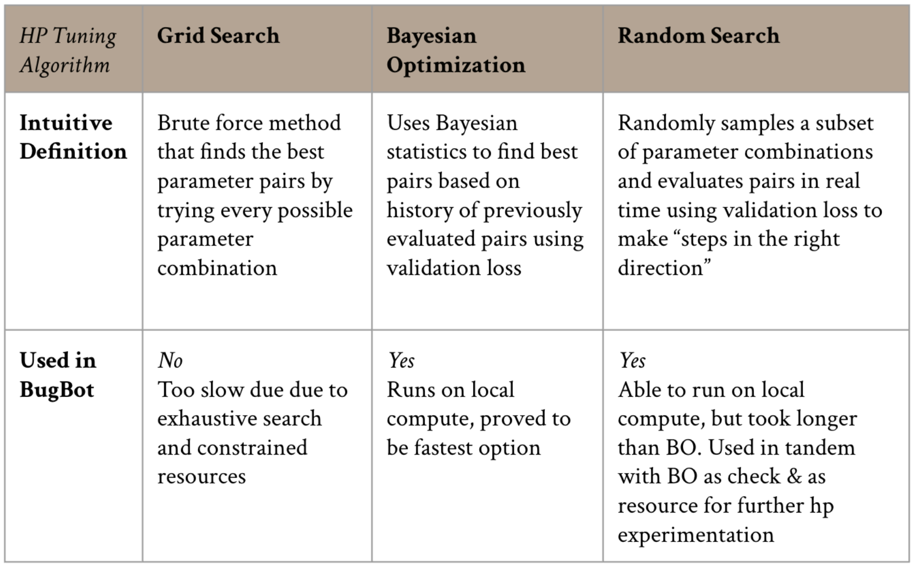
Two hyperparameter tuning methodologies were implemented for each model using Keras Tuner: Bayesian Optimization and RandomSearch. Notably, grid search was initially implemented for our models.

Inefficiency of Grid Search

Grid search was initially implemented but was quickly found to be inefficient for the project's constrained resources in terms of both compute and time. The inefficiency of grid search is due to its brute-force/exhaustive approach to exploring all possible hyperparameter combinations, resulting in high computational costs.

Efficient Algorithms Overview

In contrast, Bayesian optimization and random search explore the hyperparameter search space more efficiently by focusing on probable combinations or reviewing only a subset of combinations. Essentially, these algorithms make smarter and more efficient decisions compared to the initial grid search approach, requiring fewer trials and less computational resources to identify optimal hyperparameters. Figure 2.4: Optimization Algorithm Summary provides an intuitive summary of the three algorithms and their consideration in BugBot tuning. Details are further discussed in the following section.



HP Tuning Algorithm	**Grid Search**	**Bayesian Optimization**	**Random Search**
Intuitive Definition	Brute force method that finds the best parameter pairs by trying every possible parameter combination	Uses Bayesian statistics to find best pairs based on history of previously evaluated pairs using validation loss	Randomly samples a subset of parameter combinations and evaluates pairs in real time using validation loss to make "steps in the right direction"
Used in BugBot	*No* Too slow due due to exhaustive search and constrained resources	*Yes* Runs on local compute, proved to be fastest option	*Yes* Able to run on local compute, but took longer than BO. Used in tandem with BO as check & as resource for further hp experimentation

Figure 2.4: Optimization Algorithm Summary

In BugBot, Bayesian optimization is used as the main optimization algorithm and is specifically set to optimize combinations by minimizing validation loss. Random search is used to help verify the robustness of hyperparameters and to provide other potential hyperparameter combinations to explore if results differ from the Bayesian optimization result. Since neither algorithm explores the entire search space, setting the max_trials parameter is important. This parameter defines the maximum number of combinations the given algorithm will attempt. Keras Tuner is used to implement both algorithms and provides the max_trials parameter for easy use.

Hyperparameters

For each model, the learning rate, dropout rate, and batch size are tuned, resulting in the following hyperparameter space:

- Learning rate: [0.01, 0.001, 0.0001]
- Dropout rate: [0.2, 0.3, 0.4, 0.5]
- Batch size: [16, 32, 64]

Given three choices for learning rate, four for dropout, and three for batch size, the maximum number of trials is 36 (3 * 4 * 3). Therefore, we set max_trials to 20, representing 20/36. This means the algorithms will, at most, search about 55% of the original search space compared to grid search, which would explore all 36.

2.4.5 Hyperparameter Optimization Workflow

The hyperparameter tuning is implemented as the following process for each model:

For each algorithm (Bayesian and Random Search):

1. Multiple trials are conducted with different hyperparameter combinations via the algorithm
2. Each model is evaluated on validation data from `VAL_GENERATOR`
3. Early stopping halts training as needed, with the last epoch being saved
4. The best-performing hyperparameter combination is identified and saved to a dedicated dictionary for the current model and current optimization algorithm
5. A final model is constructed with the best parameters identified in Step 4
6. The model is trained for the optimal number of epochs identified in Step 3
7. Evaluation metrics (see Chapter 4) are calculated on both validation and test sets and saved to CSV for comparison with other models

Early Stopping Tuning

Multiple independent tuning runs were conducted to ensure the robustness of the model, particularly concerning early stopping parameters. The model tuning pipelines were tested using the argparse library for efficient parameter modification through the command line. It tested patience values in the list [3, 5, 10] and min_delta values in the list [0.0001, 0.001, 0.01].

The systematic methodology we use for BugBot combines transfer learning, custom classification heads, and dual hyperparameter optimization algorithms for robust pest identification that employs different overfitting mitigation techniques.

2.4.6 Final Testing

After DenseNet201, Xception, and MobileNetV2 completed their respective hyperparameter tuning, the final models were built using the hyperparameter tuning results for best learning rate, best batch size, and best dropout rate, which were uniquely found using the hyperparameter tuning described in the previous section for each model. Additionally, random seeds are defined as constants for reproducibility and include the following libraries, each with a seed value of 2025:

- random
- numpy
- tensorflow
- keras

Final Model History

The final models are trained on the training dataset, and the history is of training and learning is saved for both training data and validation data. This history is important because it is used to gain important insight into the model's learning patterns by tracking changes in accuracy and loss over time.

Final Model Evaluation

After the model is trained, the test data is passed through the model for classification. The results are then used to evaluate the model's performance. The metrics and diagrams reported include the following: macro-averaged one-vs-rest ROC AUC score, accuracy, precision, recall, F1 Score, classification report, confusion matrices, and ROC curve. These metrics are defined and analyzed in the following Results section.

Chapter 3

Project Demonstration

3.1 Streamlit

3.1.1 Deployment Approach

To accomplish our objectives of creating a user-friendly application and deploying our custom model, we decided to use Streamlit, an open-source app framework that allows users to turn scripts into shareable web apps. Alternatively, users can run the Streamlit app locally by visiting BugBot's GitHub repository, navigating to the webapp folder, downloading all the files, and executing the Streamlit application locally by running streamlit run app.py after installing all required dependencies. To access BugBot's Streamlit web application, users can visit https://bugbot.streamlit.app/.

3.1.2 User Interaction and Output

Users can upload an image of an insect, and the model will return the top three predicted classes along with their corresponding probabilities, indicating the model's confidence in each prediction. Figure 3.1 depicts a user flow from image upload to prediction output, and Figure 3.2 shows an example output when a user uploads an image.

3.1.3 Image Processing and Prediction

When a user uploads an image, the backend executes the following steps (as shown in Figure 3.1):

- Resized to 224×224 pixels to match the input size
- Converted from RGBA to RGB
- Normalized pixel values scaled [0, 1]
- Reshaped to a 4D tensor (1, 224, 224, 3) for model compatibility

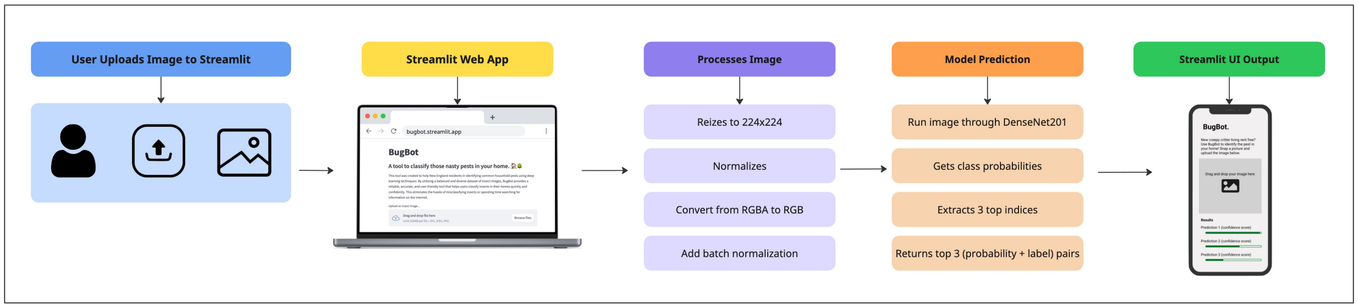
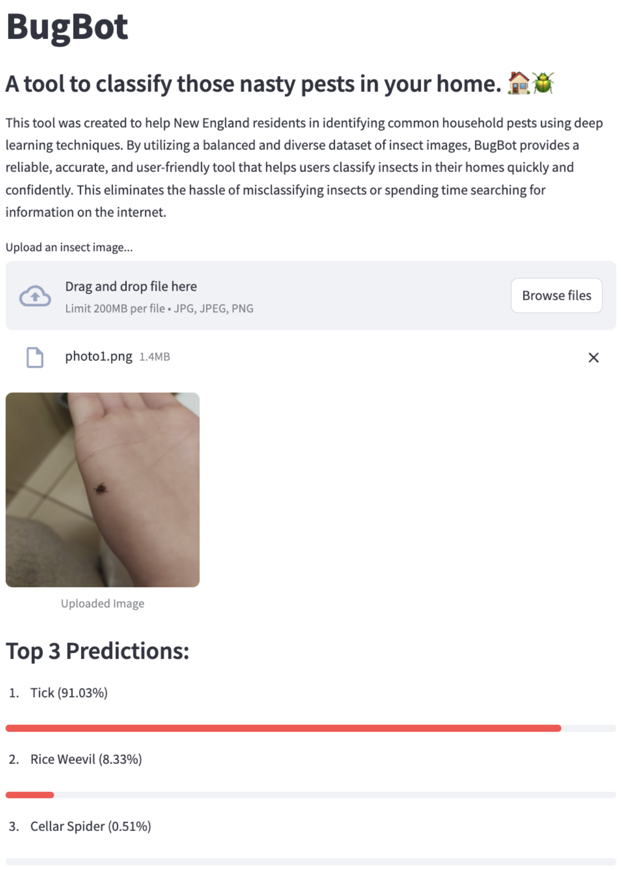


Figure 3.1: Streamlit user flow-chart



BugBot

A tool to classify those nasty pests in your home. 🏠🪲

This tool was created to help New England residents in identifying common household pests using deep learning techniques. By utilizing a balanced and diverse dataset of insect images, BugBot provides a reliable, accurate, and user-friendly tool that helps users classify insects in their homes quickly and confidently. This eliminates the hassle of misclassifying insects or spending time searching for information on the internet.

Upload an insect image...

Drag and drop file here
Limit 200MB per file • JPG, JPEG, PNG

Browse files

photo1.png 1.4MB

Uploaded Image

Top 3 Predictions:

1. Tick (91.03%)
2. Rice Weevil (8.33%)
3. Cellar Spider (0.51%)

Figure 3.2: Example of a user-uploaded image of a tick with top 3 predictions

Chapter 4

Discussion and Future Work

4.1 Evaluation Metrics

To ensure a thorough assessment of model performance and guide the selection of the final model, we used a comprehensive set of key metrics: accuracy, loss curve, confusion matrix, precision, recall, and F1 score. These are standard metrics used for classification tasks that will allow us to show performance, track model history, and see the model's ability to accurately identify pests. The final model decision was further supported by supplemental metrics, comprising receiving operating curves (ROC) and one versus rest scores (OVR), which provided additional insights into the model's discrimination ability across different threshold settings for multi-class classification. The ROC curve helps visualize the trade-off between true positive and false positive rates, while the OVR scores measure performance for each class independently, ensuring fair assessment across all pest categories. By using a diverse set of metrics, we can have a full evaluation of our models, capturing overall performance and class-specific behavior, ultimately leading to the final model selection.

- **Accuracy**: A metric to correctly detect pests. It is given by:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Precision**: A metric to determine the number of successful positive predictions the model has created. It is given by:

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall**: A metric that counts the number of accurate positive predictions made out of all possible positive predictions. It is given by:

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **F1 Score**: A metric that is the harmonic mean of the recall and precision scores, used to compare two classifiers. A score of 1 indicates a flawless classifier. It is given by:

$$F1\ Score = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \times 100$$

- **ROC AUC**: A plot of the false positive rate versus the true positive rate. The area under the curve (AUC) is used to compare different models.
- **One-versus-rest score**: A classification strategy that trains one binary classifier per class. For each classifier, the target class is treated as positive, while all other classes are treated as negative. This approach is commonly used for multiclass classification problems.
- **Macro Average**: The arithmetic mean of each class related to precision, recall, and F1 score. This evaluation is used for the overall performance of multiclass classification. It is given by:

$$\text{Macro Average} = \frac{1}{N} \sum_{i=1}^{N} \text{measure in class}_i$$

4.2 Results

This section discusses the results of our custom three models: Xception, MobileNetV2, and DenseNet201. First, we will discuss the performance based on the loss and accuracy curves, the confusion matrix, the classification report, the ROC-AUC curve, and the summary chart. Then, we will interpret and conduct an overall, comprehensive assessment using all the evaluation metrics to determine the final model.

4.2.1 Loss Curve

The training and validation loss curves for the three models are depicted in Figure 4.1. Each model was trained for 19 epochs.

Figure 4.1a shows that the DenseNet201 model has some instability in the validation loss, which is characterized by fluctuations throughout the training. The training loss is relatively stable; however, since the validation loss fluctuates, this suggests some overfitting.

Figure 4.1b shows that MobileNetV2 has a consistent decrease in training loss over time, with the validation loss remaining relatively stable across epochs.

Although there is a gap between training and validation loss, the validation curve shows minimal fluctuation and no signs of severe overfitting.

Similar to MobileNetV2, figure 4.1c shows that Xception shows a decline in training loss and a stable validation loss curve. While there is a noticeable gap between the two losses, the validation loss remains low and consistent throughout training.

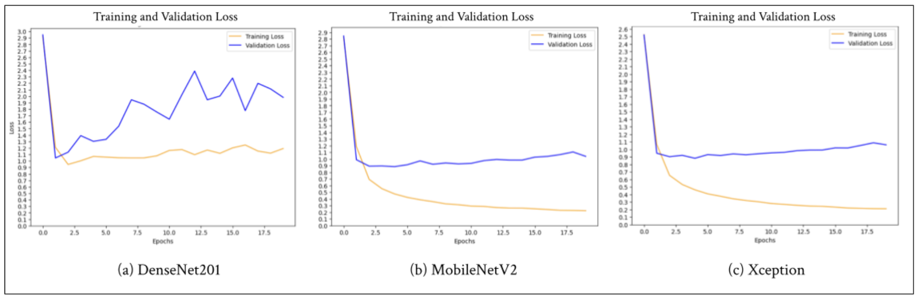


Figure 4.1: Loss curve (a) DenseNet201; (b) MobileNetV2; (c) Xception

4.2.2 Accuracy Curve

The training and validation accuracy curves for the three models are depicted in Figure 4.2.

Figure 4.2a shows that DenseNet201 achieves consistent and relatively high validation accuracy throughout training, with both training and validation accuracies stabilizing around 80%. The gap between training and validation accuracy is minimal, and the validation curve does not exhibit extreme fluctuations. These results suggest good generalization and minimal overfitting despite the instability observed in the loss curves.

Figure 4.2b shows that MobileNetV2 shows a steady increase in training accuracy, reaching over 90% by the end of training. However, the validation accuracy plateaus around 75% early in training and fluctuates slightly without significant improvement. The gap between training and validation accuracy indicates signs of overfitting.

Xception achieves the highest training accuracy among the three, nearing 95% by the final epoch, as shown in figure 4.2c. However, similar to MobileNetV2, the validation accuracy remains consistently around 73–75% with no upward trend. The gap between the training and validation accuracy suggests overfitting.

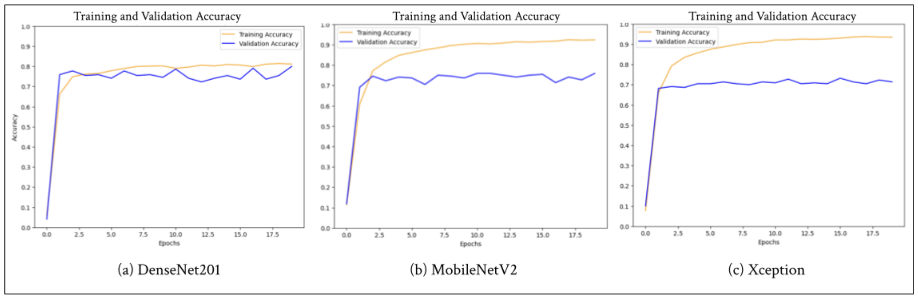


Figure 4.2: Accuracy curve (a) DenseNet201; (b) MobileNetV2; (c) Xception

4.2.3 Confusion Matrix

The confusion matrix for the three models is depicted in Figure 4.3.

In Figure 4.3a, DenseNet201 exhibits the strongest overall performance among the three models. Most classes show high true positive counts along the diagonal, indicating accurate classification. Notably:

- Cellar spider, American house spider, and house centipede have high correct classifications (20, 20, and 18 correct predictions)
- The highest amount of misclassifications is observed between bedbug, tick, and flea, which may be due to the visual similarities of the insects

In Figure 4.3b, MobileNetV2 demonstrates slightly reduced performance in comparison to DenseNet201 but still achieves relatively high accuracy:

- Carpenter Ant, cellar spider, and house centipede maintain strong classification accuracy (16, 20, and 18 correct predictions)
- Confusion increases among bedbug, flea, and tick, with more misclassifications distributed across these classes compared to the DenseNet201 model
- Notably, rice weevil, flea, and silverfish show higher error rates, suggesting the model may be less adept at learning features for these classes

Figure 4.3c shows more misclassifications across the majority of the insects for Xception.

- The highest amount of correct classifications is for the American house spider, cellar spider, and house centipede (16, 18, 18)
- However, Xception was able to have the lowest misclassifications for bedbugs

- Subterranean termite was getting misclassified as the American house spider, bedbug, carpenter ant, flea, rice weevil, and silverfish.

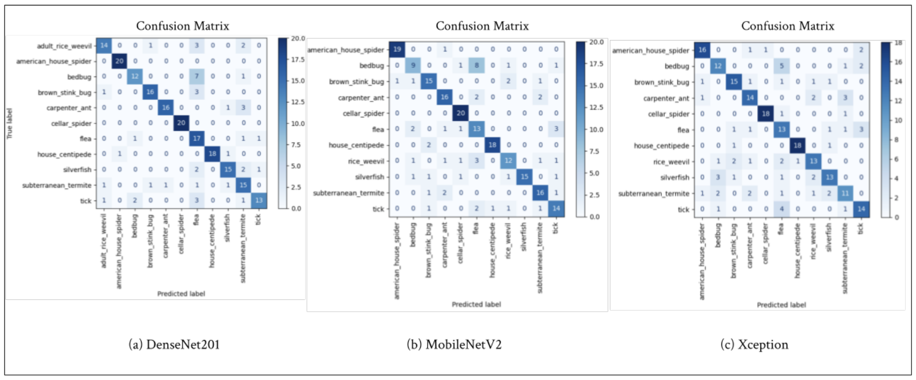


Figure 4.3: Confusion matrices (a) DenseNet201; (b) MobileNetV2; (c) Xception

4.2.4 Classification Report

This section provides a comparative evaluation of the three based on their classification performance across 11 insect classes. The performance metrics analyzed include precision, recall, and F1-score.

DenseNet201 showed balanced precision and recall across most classes, as shown in Table 4.1, with particularly strong results for cellar spider (F1-score: 100%), house centipede (95%), and American house spider (98%).

MobileNetV2 also excelled in detecting cellar spider and house centipede on Table 4.2, but exhibited reduced precision in classes such as bedbug (64%), flea (46%), and rice weevil (71%).

Xception retained high precision and recall for cellar spider and house centipede, similar to the other models; it experienced more significant drops in performance for classes as shown in Table 4.3, like subterranean termite (F1-score: 58%), flea (54%), and bedbug (62%). These metrics suggest the model may be more sensitive to inter-class variability or feature ambiguity in the dataset.

All three models consistently performed well on visually distinctive classes such as cellar spider, house centipede, and American house spider, suggesting that these insects have distinctive features that are captured during training. Conversely, classes such as flea, bedbug, and subterranean termite remained challenging across models, indicating potential issues with detecting features, image diversity, or quality.

Class	Precision	Recall	F1-score	Support
American House Spider	0.95	1.00	0.98	20
Bedbug	0.80	0.60	0.69	20
Brown Stink Bug	0.89	0.80	0.84	20
Carpenter Ant	0.94	0.80	0.86	20
Cellar Spider	1.00	1.00	1.00	20
Flea	0.47	0.85	0.61	20
House Centipede	1.00	0.90	0.95	20
Rice Weevil	0.82	0.70	0.76	20
Silverfish	0.83	0.75	0.79	20
Subterranean Termite	0.60	0.75	0.67	20
Tick	0.87	0.65	0.74	20

Table 4.1: DenseNet201 classification report

Class	Precision	Recall	F1-score	Support
American House Spider	0.95	0.95	0.95	20
Bedbug	0.64	0.45	0.53	20
Brown Stink Bug	0.75	0.75	0.75	20
Carpenter Ant	0.76	0.80	0.78	20
Cellar Spider	0.83	1.00	0.91	20
Flea	0.46	0.65	0.54	20
House Centipede	0.95	0.90	0.92	20
Rice Weevil	0.71	0.60	0.65	20
Silverfish	1.00	0.75	0.86	20
Subterranean Termite	0.76	0.80	0.78	20
Tick	0.67	0.70	0.68	20

Table 4.2: MobileNetV2 classification report

Class	Precision	Recall	F1-score	Support
American House Spider	0.84	0.80	0.82	20
Bedbug	0.63	0.60	0.62	20
Brown Stink Bug	0.75	0.75	0.75	20
Carpenter Ant	0.70	0.70	0.70	20
Cellar Spider	0.95	0.90	0.92	20
Flea	0.46	0.65	0.54	20
House Centipede	0.95	0.90	0.92	20
Rice Weevil	0.68	0.65	0.67	20
Silverfish	0.72	0.65	0.68	20
Subterranean Termite	0.61	0.55	0.58	20
Tick	0.67	0.70	0.68	20

Table 4.3: Xception classification report

4.2.5 ROC-AUC Curve

Figure 4.4 shows ROC curves and corresponding AUC values for a multi-class classification task involving 11 classes (Class 0 through Class 10). The ROC-AUC curves compare the classification performance of the models across the different pest classes.

In Figure 4.4a, DenseNet201 has the most tightly clustered ROC curves near the top-left corner, signifying consistently strong classification across all classes. Some key observations:

- Perfect classification is achieved for Class 1 and Class 5 (AUC = 1.00)
- The lowest AUC is observed for Class 0 (AUC = 0.94)
- The average AUC across all classes remains high, indicating robust model performance

MobileNetV2 also exhibits strong performance with AUC values ranging from 0.91 to 1.00, as shown in Figure 4.4b. Key observations include:

- Perfect classification is achieved for Class 0 and Class 4 (AUC = 1.00)
- Slightly lower AUCs are observed for Classes 5, 8, and 9 (AUC = 0.91-0.93), suggesting these classes are relatively harder to classify
- All AUC scores are over 90, indicating strong classification and separability performance

Xception also has high classification effectiveness, with AUC values between 0.91 and 1.00, as shown in Figure 4.4c. Notable findings include:

- Class 4 achieves perfect classification (AUC = 1.00),
- The lowest AUC values are found in Class 5 (AUC = 0.91) and Class 7 (AUC = 0.92)
- Other classes, such as Class 0 and Class 6, attain high AUCs near 0.99 and 0.98, reflecting strong discriminative ability

4.2.6 Summary Chart

Table 4.4 shows a comparative evaluation of Xception, MobileNetV2, and DenseNet201 based on five key performance metrics: precision, recall, F1 score, accuracy, and OVR score.

Among the models, DenseNet201 achieves the highest performance across all metrics, with a precision of 83%, recall of 80%, f1 score of 81%, accuracy of 80%, and a OVR score of 97%. MobileNetV2 follows with moderately strong results, recording values of 78% for precision, 76% for recall, and an overall accuracy

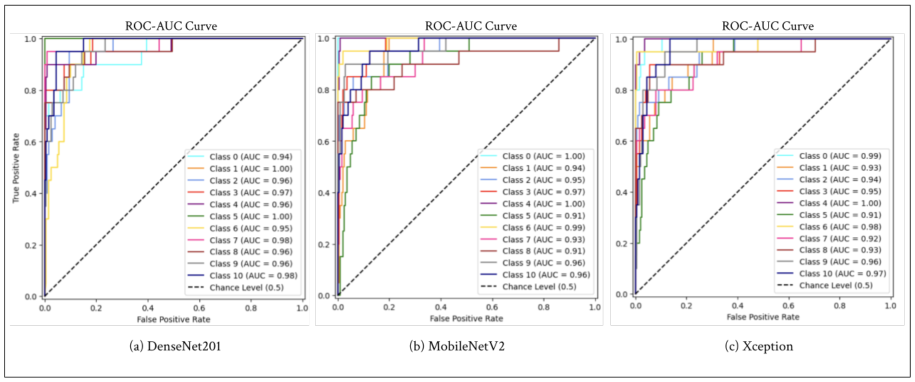


Figure 4.4: ROC-AUC curve (a) DenseNet201; (b) MobileNetV2; (c) Xception

and F1 score of 76%, along with an OVR score of 96%. Xception exhibits the lowest performance among the three, with precision, recall, and accuracy values around 71-72%, and a OVR score of 95%. These findings indicate that while all models perform reasonably well, DenseNet201 offers the best results across all the metrics, making it the most effective model in this evaluation.

Model Name	**Precision (%)**	**Recall (%)**	**F1 Score (%)**	**Accuracy (%)**	**OVR Score (%)**
Xception	72	71	72	71	95
MobileNetV2	78	76	76	76	96
DenseNet201	83	80	81	80	97

Table 4.4: Performance comparison of different models

4.3 Interpretation

Loss and Accuracy Curves

For the loss and accuracy curves, although DenseNet201 had fluctuations for the validation loss, there was little to no gap between the training and validation accuracy. Whereas Xception and MobileNetV2 had a large gap between the training and validation accuracy, which means the models were overfitting to the training data and struggled to generalize well to unseen images.

Classification Reports
The classification reports revealed that Xception wasn't as consistent compared to DenseNet201 and MobileNetV2. Although both DenseNet201 and MobileNetV2 attained a precision of 100% for at least one insect class, Xception did not achieve perfect precision for any class.

Confusion Matrices
For the confusion matrix, all models performed moderately well, however, there were some notable observations:

- DenseNet201 had the most accurate and reliable model, with the clearest class separations and the fewest misclassification compared to Xception and MobileNetV2
- MobileNetV2, though efficient and fast, had multiple class overlaps, making it more suitable for situations where computational constraints outweigh the need for high precision
- Xception provided a reasonable balance between model complexity and classification accuracy but showed moderate confusion in several classes

ROC-AUC Curves
All three models exhibit excellent overall performance with AUCs well above the chance level (0.5) across all classes. Among them:

- DenseNet201 shows the most consistent and high AUCs, making it a strong candidate since classification accuracy is the main priority
- MobileNetV2, while slightly less consistent, performed well (achieving a 96% OVR score) and runs faster compared to DenseNet201 due to fewer parameters and an architecture built for mobile applications
- Xception offers a balanced alternative with high AUCs, although it has some performance drops in select classes

Summary Table
The summary table, as shown in Table 4.4, highlighted how DenseNet201 consistently outperforms the other models across all metrics.

Overall, DenseNet201 leads in performance consistency, while MobileNetV2 and Xception are viable alternatives based on computational constraints. Therefore, based on a mixture of attaining high accuracy, classification, and computational resources, DenseNet201 was selected as our final model, which will be used in our Streamlit web application.

4.4 Future Work

To address the current limitations, future work could focus on expanding the dataset by collecting more diverse images of household pests. Implementing additional data augmentation techniques could further enhance model robustness. Additionally, exploring more advanced deep learning architectures that balance performance and computational efficiency could improve classification accuracy without sacrificing accessibility.

4.5 Bias Mitigation Strategies

While data augmentation played a key role in improving model robustness, further measures are necessary to detect and mitigate biases that can affect both fairness and generalizability in real-world applications.

4.5.1 Class-wise Error Analysis

In addition to overall accuracy, examining class-specific performance metrics is essential for uncovering latent model biases. In our confusion matrix analysis, certain species such as fleas and bed bugs were frequently misclassified due to visual similarity. Future work should involve tracking false positive and false negative rates per class to better understand the classifier's strengths and weaknesses and prioritize augmentation or resampling accordingly.

4.5.2 User Upload Bias

Since our system is intended for user-uploaded images (e.g., via smartphone), we must consider variability in image quality, lighting, and framing. These inconsistencies can introduce systematic biases, particularly if users tend to upload similar types of images (e.g., blurry close-ups). Mitigation strategies include:

- Image quality control: Preprocessing can be extended to assess image clarity and completeness before classification.
- Duplicate detection: Image hashing and metadata analysis can flag and down-weight repeated uploads or near-duplicates.
- User feedback loop: Incorporating user-confirmed corrections (e.g., "Was this classification correct?") enables active learning and long-term improvement of the model.

4.5.3 Long-term Class Imbalance Monitoring

Although the initial dataset is balanced, real-world data collection may not be. A monitoring system should be established to detect disproportionate representation of certain pest types and, if needed, reweight the training loss or apply class-specific augmentation strategies.

Chapter 5

Conclusion

5.1 Key Takeaways

Throughout this project, we have learned how much diverse training data we need to train the model. Moreover, implementing model tuning and experimenting with different hyperparameters, such as batch size and epochs, can make a significant difference in terms of improving metrics. This led to the team discovering that sometimes a simpler architecture can work better than something complex.

5.2 Broader Project Objectives

Currently, there is a strong emphasis on image classification for agricultural pests, therefore, our research aims to expand this focus to pests commonly found in homes. By developing an accessible and efficient pest identification model, BugBot aligns with the broader objective of providing residents with a tool for identifying household pests using smartphone images. Our results contribute to this goal in several ways:

- Improved Classification for Real-World Images: Unlike many existing datasets that rely on professionally captured insect images, BugBot's model is trained on a dataset curated from user-uploaded images. This ensures that our classifier is robust to variations in lighting, angle, and image quality, making it more applicable for everyday users.
- Insights for Pest Management: Accurate pest identification is important for informed decision-making in pest control. BugBot quickly fixes this issue by providing instant feedback on potential infestations.
- Expansion of Pest Classification Research: By shifting the focus from agricultural to household pests, our work broadens the application of artificial

intelligence based in pest identification. This could encourage further research into urban pest classification, crowd sourced data collection, and pest control solutions.

The next steps in refining our approach would be to expand the dataset by collecting more diverse and high-quality images of household pests. Implementing additional data augmentation techniques could further enhance model robustness. Additionally, exploring more advanced deep learning architectures that balance performance and computational efficiency could improve classification accuracy without sacrificing accessibility.

Chapter 6

Reproducibility

6.1 GitHub

The Github repository for the BugBot project can be reached here: https://github.com/keeganveazey/BugBot

6.2 Repository Structure

Figure 6.1: GitHub Repository Structure visualizes the repository and directory structure of the BugBot project.

- DATA folder houses the raw data with manual filtering step completed
- Best Models folder contains the notebooks which build the final DenseNet201, Xception, and MobileNetV2 models with the tuned hyper parameters
- Model Tuning folder contains the scripts to run hyperparameter tuning for each model
- BugBot Architecture Diagrams folder contains diagrams relevant to BugBot
- webapp folder contains resources for Streamlit deployment
- Reports folder contains iterative reports completed throughout project process and presentations
- Files in orange are lone files and are described as follows:
  - global_bug_bot_functions.py contains evaluation functions and data splitting functions used in multiple files across the repository
  - data_preprocessing_pipeline.py is the script to run preprocessing on the provided raw data

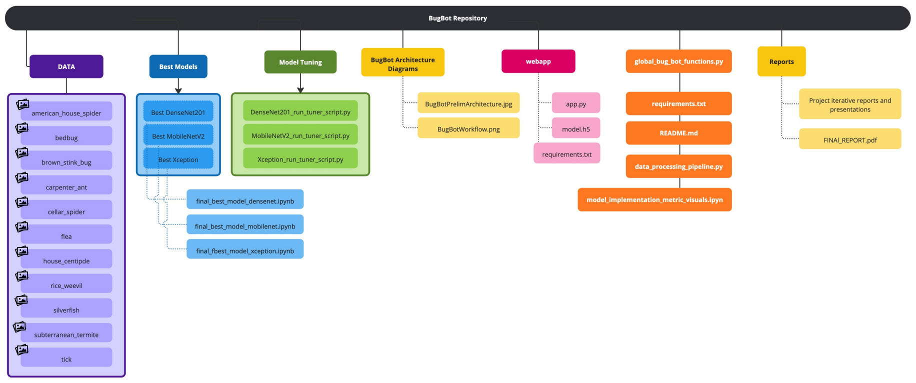

BugBot Repository
DATA
american_house_spider
bedbug
brown_stink_bug
carpenter_ant
cellar_spider
flea
house_centipde
rice_weevil
silverfish
subterranean_termite
tick
Best Models
Best DenseNet201
Best MobileNetV2
Best Xception
final_best_model_densenet.ipynb
final_best_model_mobilenet.ipynb
final_fbest_model_xception.ipynb
Model Tuning
DenseNet201_run_tuner_script.py
MobileNetV2_run_tuner_script.py
Xception_run_tuner_script.py
BugBot Architecture Diagrams
BugBotPrelimArchitecture.jpg
BugBotWorkflow.png
webapp
app.py
model.h5
requirements.txt
global_bug_bot_functions.py
requirements.txt
README.md
data_processing_pipeline.py
model_implementation_metric_visuals.ipyn
Reports
Project iterative reports and presentations
FINAl_REPORT.pdf

Figure 6.1: GitHub Repository Structure

- README.md is a file used to detail information about the repository and provides a project overview
- model_implementation_metric_visuals.ipynb is a notebook containing code used to run and measure performance of seven originally considered models from which DenseNet201, Xception, and MobileNetV2 were chosen as top performers.

6.3 Data Access

The raw dataset is available in the repository in the folder "DATA". Inside the DATA folder, there are 11 subfolders, each consisting of 1 of the 11 insect classes as shown in Figure 6.1: GitHub Repository Structure. To reproduce our data preprocessing and training, validation and test set splitting, the user can follow the instruction in the "Set Up" section of README.md:

1. Pip install requirements.txt to a virtual environment (Python 3.12)
2. cd to repo location (change directory)

Complete the following if you do not already have an environment set up the repository's requirements.txt:
Type: pip install virtualenv, press enter.

1. Update pip if needed (type: pip install –upgrade pip, press enter).
2. Type: python -m venv bugbot_env, press enter
3. Type: source bugbot_env/bin/activate, press enter
4. You should now see something that looks like (bugbot_env)(base) in front of your curser in the terminal

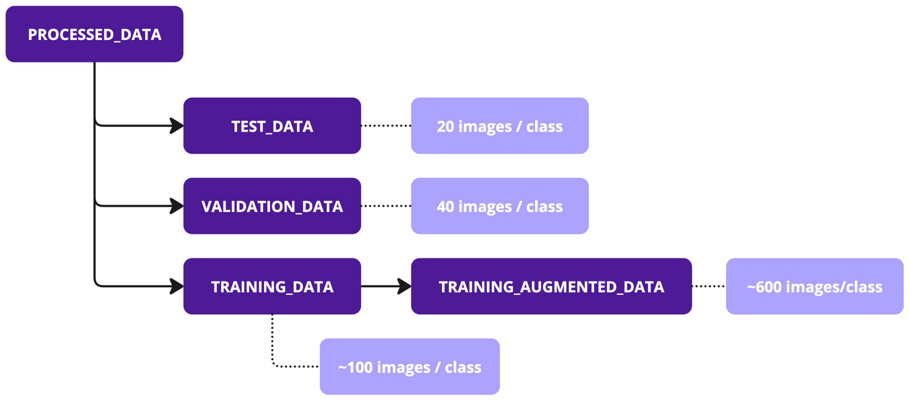


Figure 6.2: Data Preprocessing Pipeline Output Structure

5. Type: pip install -r requirements.txt, press enter

Use the preprocessing pipeline script to reproduce the model-ready dataset:

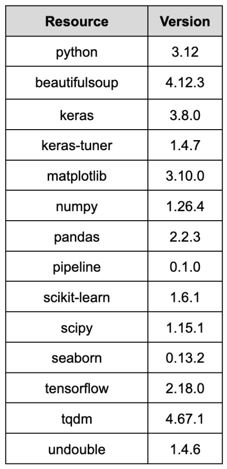
1. To run the notebook you are interested in, type jupyter notebook in the terminal of your now activated environment. After the browser opens, click the notebook of interest.

2. To run the preprocessing script, download the raw data in DATA provided in the repository. Type: python data_processing_pipeline.py

After following the above steps, the raw data will have undergone standardization and augmentation as discussed in Section 2.3: Dataset and Preprocessing and Section 2.4.2: Data Preprocessing and Data Split. The final, model-ready preprocessed data will be in the user's directory as shown in Figure 6.2: Data Preprocessing Pipeline Output Structure. The data split is done using a defined random seed so the user's script run will result in same split as the BugBot team.

6.4 Dependency Management

Notably, it is extremely important to install requirements.txt and use Python 3.12 to ensure that dependencies and versioning are taken care of before running any code. Figure 6.2: Libraries and Versioning provides the explicitly called libraries and their versions. Please refer to the requirements.txt file in the GitHub repository for a comprehensive list.



Resource	Version
python	3.12
beautifulsoup	4.12.3
keras	3.8.0
keras-tuner	1.4.7
matplotlib	3.10.0
numpy	1.26.4
pandas	2.2.3
pipeline	0.1.0
scikit-learn	1.6.1
scipy	1.15.1
seaborn	0.13.2
tensorflow	2.18.0
tqdm	4.67.1
undouble	1.4.6

Figure 6.3: Libraries and Versioning

Bibliography

Guo, B., Wang, J., Guo, M., Chen, M., Chen, Y., & Miao, Y. (2024). Overview of pest detection and recognition algorithms. *Electronics*, *13*, 3008–3008. https://doi.org/10.3390/electronics13153008

Islam, M. T., & Rahman, M. S. (2024). An efficient deep learning approach for jute pest classification using transfer learning, 1473–1478. https://doi.org/10.1109/iceeict62016.2024.10534395

Rehman, A., Naz, S., Razzak, M. I., Akram, F., & Imran, M. (2019). A deep learning-based framework for automatic brain tumors classification using transfer learning. *Circuits, Systems, and Signal Processing*, *39*, 757–775. https://doi.org/10.1007/s00034-019-01246-3

Talukder, M. S. H., Chowdhury, M. R., Sourav, M. S. U., Rakin, A. A., Shuvo, S. A., Sulaiman, R. B., Nipun, M. S., Islam, M., Islam, M. R., Islam, M. A., & Haque, Z. (2023). Jutepestdetect: An intelligent approach for jute pest identification using fine-tuned transfer learning. *Smart Agricultural Technology*, *5*. https://doi.org/10.1016/j.atech.2023.100279

Thenmozhi, K., & Srinivasulu Reddy, U. (2019). Crop pest classification based on deep convolutional neural network and transfer learning. *Computers and Electronics in Agriculture*, *164*, 104906. https://doi.org/10.1016/j.compag.2019.104906

Turkoglu, M., Yanikoğlu, B., & Hanbay, D. (2021). Plantdiseasenet: Convolutional neural network ensemble for plant disease and pest detection. *Signal, Image and Video Processing*, *16*. https://doi.org/10.1007/s11760-021-01909-2