HIDING IN PLAIN SIGHT: IDENTIFYING CRYPTIC CROP PESTS IN BRAZIL WITH AI

Samuel Mashil

SUMMER RESEARCH PROJECT 2023

DEPARTMENT OF ZOOLOGY
University of Cambridge

Table of Contents

The GitHub repository containing all the code for this project can be found here:

https://github.com/TangyPenguin37/MothBot

All the data for the project (including both the original and processed images) is here:

https://bit.ly/mothbot-data

Background

Helicoverpa armigera is one of the world's most dangerous crop pests. Originally present in just Europe and Asia, the species found its way to South America in 2013 and began hybridizing with the local Helicoverpa zea species, sharing its pesticide resistance genes.

Monitoring these two species has always been an expensive and time-consuming process for farmers, involving sending samples to labs or universities for genotyping – a process only getting less reliable with further hybridization. Over the course of this project, I intended to create a machine learning model that could reliably distinguish the two species using no more than images of the moths' wings.

The Processing Pipeline

Data Gathering

Unsurprisingly, the first step of the process involved gathering images of several hundred moths' wings. This step was already done for me by a school student volunteer, before a fellow summer internship student took further pictures, giving me exactly 1800 pictures to work with. Each picture looked as follows, with a ruler, ColorChecker, specimen ID, and 4 quadrants for each of the 4 wings (although some individuals did have missing wings):

Each individual moth had two photos for each of the dorsal and ventral sides, equating to 900 individual moths to work with. I was also given a sequencing spreadsheet, giving the admixture proportions for each specimen (where an admixture between 5% and 95% was classed as a hybrid).

The next step was to process all these photos to get the data I desired.

Extracting features from images

Originally, I used a mix of macros in ImageJ (specifically Fiji) along with Python to process all the images. I ultimately switched this entire process to Python, such that all stages of the project could be easily reusable/replicable in a single language, but the process remained otherwise identical.

Using ImageJ, I started off by cropping the images into 4, giving an image for each wing. I then converted each of the resulting RGB (Red, Green, Blue) photos into HSB (Hue, Saturation, Brightness), and selected just the hue channel. With a hue threshold of 45 (out of 255), I was then able to create a convincing mask for the shape of the wing, from which I could measure the desired features. The only exceptions to these were shape and colour, for which ImageJ did not have functionality. Instead, I used the *pyefd* and *scikit-learn* Python packages instead. Specifically, I converted the wings' shape into a numerical representation by expressing it as a sum of ellipses, using a technique known as elliptical Fourier descriptors. For colour, I clustered all the pixel RGB values into 2 clusters using the k-means clustering algorithm, returning the central/mean colour value of each cluster, effectively giving the 2 'most dominant' colours in each wing. By the end, I had extracted the following features:

- Area
- Major/minor length, where ImageJ would fit a single ellipse to the wing and measure the 2 diameters of said ellipse.
- Aspect ratio, defined simply as $major\ axis \div minor\ axis$
- Circularity, defined as $\frac{4\pi \times area}{perimeter^2}$ and bound between 0 and 1.
- Feret diameter, defined as the maximum possible calliper measurement.

- Minimum Feret diameter, defined as the minimum possible calliper measurement.
- Shape, using the aforementioned EFD technique – giving the normalised results for an EFD of order 7.
- Colour, returning 2 RGB colour values using the k-means clustering algorithm as previously described.

Counting major and minor length separately, each of the features above yielded one numerical value, apart from shape, which gave 25 values; and colour, which gave 8 values. The 8 values for colour came from 2 colour clusters, with a red, green, blue and percentage value for each (where percentage represented the proportion of the wing in each cluster). The 25 shape values came from 7 ellipses (i.e., a 7th order EFD), each with 4 values to represent it, minus 3 standardised values because of normalisation – normalising the EFD values to be size invariant means the first three values for the first ellipse will always be 1, 0 and 0 respectively, so are therefore redundant. I also included three binary values to identify the samples by location (front or rear), wing (left or right), and side (dorsal or ventral).

Accounting for missing wings, I now had 4484 samples, each with 43 input variables. The next step was to feed all this gathered data into different potential machine learning models and observe the results.

Training and Testing

Logistic Regression

As an initial proof of concept, I started off using a logistic regression model without hybrids to prove that the idea of this project would work, before including hybrids to see the effect this would cause.

Overview

The logistic regression model is based on the following equation:

$$p = \frac{1}{1 + e^{-(a_0 + a_1 X_1 + a_2 X_2 + \cdots)}}$$

X_i represents the i-th input variable/feature, and a_i represents the weight for said feature, along with an overall bias value, a_0. For a given set of inputs, the function returns a value between 0 and 1, representing the model's prediction for the probability of the sample belonging to the '1' class rather than the '0' class. A value below 0.5 means the model predicts the '0' class, whilst a value above 0.5 suggests the '1' class, and the distance from 0.5 represents the model's confidence in said prediction.

Training such a model involves using all the training data together and adjusting the a_i weights such that the overall error (defined by a cost/loss function) across all the samples is minimized. The optimal weights cannot be calculated directly, so a solver algorithm is used to find the required weights for this minimum value. Once the most optimal weights are found, they are frozen in place, such that the model can be used for classifying new, unseen inputs.

Findings

In this case, training the model was as simple as using the scikit-learn package in Python. I started by splitting the data into training and testing sets, so the testing set could be used to measure the accuracy of the models on unseen data. I chose an 80/20 split for testing and training respectively. Whilst this split was random at first, I quickly switched to a stratified split (matching the species balances in the subsets with that of the full dataset) to reduce the large variation in accuracy that became apparent.

Additionally, during the first two weeks of the project, before the full sequencing was complete, I only had access to the species information for about 70 individuals – 35 armigera and 35 zea, so I made use of K-fold cross-validation (specifically with 5 folds) to reduce the further variation in accuracy caused by such a limited dataset, although this wasn't a worry later in the project.

An Issue: Pseudoreplication

My initial tests quickly made the issue of pseudo-replication apparent – for an individual moth, there were 8 different samples which were not independent of one another, making the accuracy figures somewhat less reliable. Additionally, the model was using the indicator variables for the wings/side of the wing as features to distinguish between species, although they should obviously have no different effects between species.

The optimal solution would be to group all data points from a single individual into a single sample, however this caused an 8x drop in the number of total data points (or more, accounting for missing wings), meaning the model suffered as a result. I ended up finding a compromise, by which I grouped the data points by wing – the dorsal and ventral side for each wing were classed as one data point, reducing the dataset by just a factor of 2. I coupled this with better splitting into training/testing subsets, to ensure all the wings from a single individual would be in either one or the other set, not split across both. This ensured that the data in the testing set was in no way dependent on data the model had already seen, and although not a perfect solution, did make the accuracy readings somewhat more reliable. This was the system I stuck with for all future models that I made and tested.

With the initial proof-of-concept tests, excluding hybrids, I reached an accuracy of 86.83%, with a standard deviation of 1.01. Including hybrids (as was my goal for the final model) meant I had to use multiclass logistic regression rather than simple binary logistic regression (as I was now working with 3 classes), giving an unsurprisingly lower, but still solid 76.94% accuracy, with standard deviation 2.92. My next idea was simply to repeat the same tests, this time using a linear discriminant analysis model.

Linear Discriminant Analysis

Overview

Linear discriminant analysis (LDA) works similarly to logistic regression in that it uses a weighted sum/linear combination of the input variables (i.e., an equation of the form $a_1X_1 + a_2X_2 + a_3X_3 + \cdots$). In this case, however, rather than using the logistic function to map the linear combination to a value between 0 and 1, the weights are calculated such that the output values for samples in separate classes are separated as much as possible. A simple decision boundary can then be drawn to categorize the model's predictions into one of two possible classes.

Once again, a solver is used alongside the training data to find the most optimal weights and cutoff value to minimize the error, before said values are fixed in place to produce a completed model.

Results

Creating the linear discriminant analysis model was as simple as a single line change from the code I already had, as all other steps remained identical and the LDA was also available through the scikit-learn package. Repeating the tests both excluding and including hybrids gave the following results:

- Without hybrids: 89.55% ± 1.29
- With hybrids: 77.82% ± 1.37

These results slightly eclipsed the logistic regression model and seemed to show the existence of some pattern in the data. Before making use of neural networks to fully realise the potential in such patterns, I wished to quickly try random forests as another drop-in method in scikit-learn, to see its performance.

Random Forests

Overview

Random decision forests make use of decision trees, where at each 'branch', a path is chosen based on the value of a single input variable. This branching process is repeated multiple times until a final

decision. However, rather than making use of a single decision tree, a random forest creates several random decision trees, and takes the consensus output of all these trees as the entire model's output.

Results

By once again simply changing one line in my code, I could implement the random forest model, giving the following results:

- Excluding hybrids: 87.21% ± 0.89
- Including hybrids: 77.06% ± 1.04

This slotted the model between the logistic regression and LDA models, although the LDA model remained the best so far. With this in mind, I finally decided to move on to neural networks, to see if they could extract further hidden patterns in the data.

Feed-forward Neural Network

Overview

Feed-forward neural networks once again make use of weighted sums of input variables but do so multiple times. A network starts with an input layer with a number of nodes corresponding to the number of input variables, followed by a series of hidden layers with varying numbers of nodes in each layer, and a final output layer with several nodes corresponding to the number of potential outputs (3 in this case). The architecture of the hidden layers can be arbitrarily chosen.

At every node (besides those in the input layer), a weighted sum of all nodes in the previous layer is taken (plus a bias), and this value is then run through an activation function to introduce an element of non-linearity into the output. This repeated process of weighted sums and activation functions enables more complex patterns in the data to be found and more complicated functions to be formed.

A process known as backpropagation is used to determine the best way to slowly optimize the weights and biases across all the training data over multiple iterations ('epochs'), and as before, the values are frozen once an optimal solution is found. This solution may not necessarily be the best, but rather is a solution where any infinitesimal change to the weights/biases will increase the cost/loss function.

Results

I started off by simply creating a neural network to make use of the same extracted features I had used till this point. I arbitrarily tried several different architectures and found that a neural network with a single hidden layer of 10 nodes, each using the ReLU activation function, worked best. Neural network functionality was not built into scikit-learn, and thus I switched to using Tensorflow in Python instead. Additionally, rather than simply splitting the dataset into training and testing subsets, I now had to include an additional validation subset. This was to estimate the performance of the model after each epoch to tell whether the improvements being made were general across the entire dataset or specific to the training subset (i.e., overfitting), and change the different hyperparameters accordingly. The testing set cannot be used for this purpose, as the final test must be completely unbiased to ensure reliability (i.e., the testing set itself must have had no impact on the actual configuration of the model).

With all this in place, I tested the neural network to reach an accuracy of 76.43% ± 2.31 (including hybrids). Whilst worse than both the random forest and LDA models, the network wasn't too far behind, and I knew further optimisations could be made. Before attempting this, however, I wished to try an image-based neural network using the images directly, and so I moved onto that next.

Convolutional Neural Network

Overview

Convolutional neural networks are a form of feed-forward neural network specifically designed to use images as inputs. They differ in that the input and output to each layer represents a multidimensional matrix (i.e., a tensor) of values. Matrix operations can be performed on the input matrices in combination with filter/kernel matrices, to reduce an input tensor down to matrices of more manageable sizes, without the sheer number of weights and biases that would be required for an equivalent typical feed-forward network. At some point, however, the matrix must be flattened and sent through at least one fully connected layer, such that a one-dimensional output can produced.

There are several possible layers in a convolutional neural network, described as follows:

Dense/Fully Connected Layers

These are identical to normal hidden layers in a typical one-dimensional feed-forward network, with a given number of nodes for which the weights and biases must be determined through training.

Normalisation Layers

These layers simply normalise the input values to have a mean of 0 and a standard deviation of 1.

Rescaling Layers

These layers simply rescale the input values to a given range by dividing them (e.g., dividing all the potential RGB colour values by 255 from range 0-255 to 0-1)

Dropout Layers

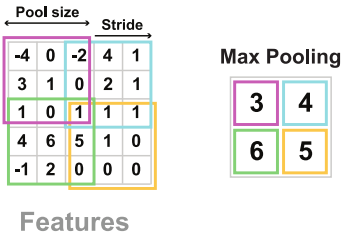
Also found in conventional feed-forward networks, these layers simply set random input values to 0 at a given rate to help prevent potential overfitting.

Flatten Layers

This layer simply flattens the multidimensional tensor input into a single one-dimensional set of values, such that it can be run through the usual fully connected and dropout layers.

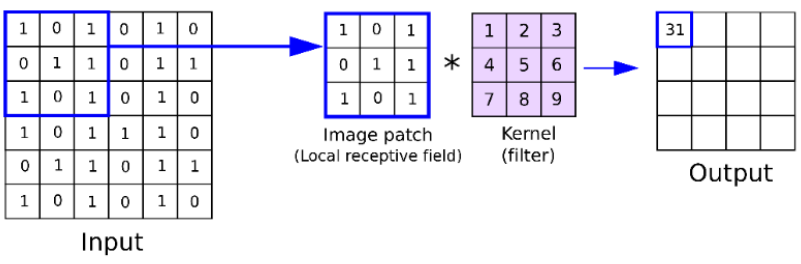
MaxPooling Layers

The pooling layers reduce the size of the input matrix by subdividing it into overlapping regions of a given 'pool' size that are separated in each dimension by a given 'stride' value, and taking the maximum value of each region, as shown here for a 3 x 3 pool and a stride of 2.



Convolution Layers

The main type of layer in convolutional neural networks, these have a set of trainable filters/kernels, each of a fixed kernel matrix size. The process undertaken here works similarly to the pooling layers, although rather than taking the maximum value of each region, the dot product of the kernel matrix and region of the input matrix is taken instead. This is repeated for each filter, and the results are stacked together to form the output matrices. The values of the output matrices are then passed through an activation function (ReLU in this case) to introduce non-linearity. This concept of matrix multiplication is shown here.



Results

Once again, I tried several different architectures for the convolutional network, eventually settling on the following model:

Layer	Type	Details	Input Shape	Output Shape
1	Rescaling	Scales RGB values from 0-255 to 0-1	(256, 256, 3)	(256, 256, 3)
2	Convolution	96 filters, 11 x 11 kernel, stride 4	(256, 256, 3)	(62, 62, 96)
3	Normalisation	-	(62, 62, 96)	(62, 62, 96)
4	MaxPooling	3 x 3 pool, stride 2	(62, 62, 96)	(30, 30, 96)
5	Convolution	256 filters, 5 x 5 kernel, stride 1	(30, 30, 96)	(26, 26, 256)
6	Normalisation	-	(26, 26, 256)	(26, 26, 256)
7	MaxPooling	3 x 3 pool, stride 2	(26, 26, 256)	(12, 12, 256)
8	Convolution	384 filters, 3 x 3 kernel, stride 1	(12, 12, 256)	(10, 10, 384)
9	Normalisation	-	(10, 10, 384)	(10, 10, 384)
10	Convolution	384 filters, 3 x 3 kernel, stride 1	(10, 10, 384)	(8, 8, 384)
11	Normalisation	-	(8, 8, 384)	(8, 8, 384)
12	Convolution	256 filters, 3 x 3 kernel, stride 1	(8, 8, 384)	(6, 6, 256)
13	Normalisation	-	(6, 6, 256)	(6, 6, 256)
14	MaxPooling	3 x 3 pool, stride 2	(6, 6, 256)	(2, 2, 256)
15	Flatten	-	(2, 2, 256)	1024
16	Dense	4096 nodes	1024	4096
17	Dropout	Rate: 0.5	4096	4096
18	Dense	4096 nodes	4096	4096
19	Dropout	Rate: 0.5	4096	4096
20	Dense	3 nodes	4096	3

As can be seen from layer 1 above, the input images had to be of size 256 x 256, as the network required for larger image resolutions would take too long to train. This meant resizing the images, which I achieved in Python. I had two different methods in mind – first, I simply cropped each image to a bounding rectangle around the wing, and then resized it to 256 x 256. This had the disadvantage of removing information about the shape of the wing due to distortion caused. The other method I tried was to pad the rectangular image with black pixels to make it square, and then resize it. This kept the shape of the wing intact but meant less pixels were dedicated to the wing itself, potentially removing important details. Both methods, however, suffered, from the removal of information regarding wing size, along with information about which wing/side of the wing the image belonged to. I settled on using the second option, as I wished to retain information about the shape of the wing and set the background pixels to black to ensure the model would not attempt to use the background as a feature. Due to the missing size information, however, I didn't expect the model to perform as well.

This is exactly what I found - after training the model many times and taking an average, I found an accuracy of 71.90% ± 2.15, far worse than all the other methods attempted so far. Whilst the removal of certain key information was definitely playing a role here, I wished to try a pretrained network and see how much of the lost performance was due to my specific implementation.

Pre-trained Convolutional Neural Network

Overview

Pre-trained convolutional neural networks are large networks that have already been trained on millions of images. The theory is that, with enough images of random items, they can mimic the way humans and other animal distinguish and identify objects. Such models, such as the VGG16 model I chose for this project, can be downloaded and fine-tuned to fit the specific use-case.

The VGG16 model was originally designed and trained with the ImageNet database of images - a collection of over 14 million images belonging to over 20,000 categories. Specifically, it was designed for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), an annual machine learning competition using a subset of 1,000 categories from the full ImageNet dataset. With this in mind, the model had to be tweaked for my specific use case, as I was only using 3 categories. TensorFlow conveniently provides a function to do this – removing the last three layers of the model, such that one can add their own layers instead. I therefore added a flattening layer, two ReLU dense layers of 4096 and 1072 nodes respectively, a dropout layer of rate 0.2, and the final layer of 3 output nodes.

Results

Training the model was surprisingly quick, given that most of the weights and biases are fixed in place – pretraining meant that only the new final layers needed training. Training the model ended up giving an accuracy of 72.61% every single time. Whilst better than my own convolutional neural network, it was still not as good as the other methods I had tried, suggesting that the loss of information from cropping and resizing the images was a significant factor in the convolutional models' poor performance. This is something that could have been improved upon, but given the project time constraints, I moved back to the extracted features neural network, to attempt some optimisations.

Optimisations

There were a few potential optimisations I could make to the extracted features network, as described:

Overview

Early stopping

A neural network model is typically trained for a set number of epochs, and this number is a hyperparameter that can be tuned by the user. However, early stopping makes use of the validation subset to decide when enough epochs have passed. Specifically, it involves monitoring the error/loss on the validation set after each epoch and stops training once this error starts increasing. Training beyond this point would cause the validation loss to increase further, meaning the model is overfitting to specific features of the training set, and rather than generalising to the validation set or other unseen data. Avoiding this ensures the model is trained for the optimal number of epochs.

Learning rate scheduler

The learning rate dictates how much the weights and biases are changed after each epoch, and this can again be tuned by the user. A higher learning rate means the model may overshoot the optimal values (but could find said values quicker or find an even more optimal set of values), whereas a lower learning rate requires more epochs to train but will not overshoot a locally optimal set of values. Typically, this rate is fixed throughout training, but a scheduler can be used to change the learning rate throughout training, lowering it towards the end of training. This means the model can find the optimal values quickly, and then fine-tune them to be even more optimal. Whilst any scheduler function could be used, I chose the ReduceLROnPlateau callback in TensorFlow, reducing the learning rate by a factor of 3 after 50 epochs of no improvement in the validation loss.

Dropout

As previously mentioned, dropout is a technique to prevent overfitting, where random input values are set to zero at a given rate when training. The thinking is that there may be mistakes made in previous layers of the model, which are 'neutralised' and hidden by the following layers in a way that is specific to the training subset. By setting some input values to zero, the mistakes or subsequent 'corrections' may become more apparent and can then be resolved accordingly.

Regularization

Another technique used to prevent overfitting, regularization works by adding a 'penalty' term to the error function that is dependent on the weights and biases of the model. This term increases with higher weights and biases, and so the model will attempt to find an optimal solution that keeps the weights and biases as low as possible. The idea is that an overfitted model will have very high weights and biases, and so the penalty term will limit this. The two common types of regularization are L1 and L2 regularization, which use different penalty terms, and I tried both to see which would work best.

Normalisation

Certain input features (such as area or major/minor axes length) can hold any value from 0 to infinity and this large range may make it harder for the model to find patterns. Normalisation rescales these values to a smaller range (typically 0-1) to aid with this. In this case, I rescaled all input features that were not already in the range 0-1 to this range, using the MinMaxScaler function in scikit-learn.

Different architectures

This technique simply involves changing the number of nodes in the hidden layers, and the number of hidden layers themselves. This step had already been performed in the previous section, and the best architecture found then was kept for this section.

Gradient clipping

The backpropagation algorithm calculates how much the weights and biases should be changed after each training epoch. Specifically, it finds the gradient of the loss function with respect to each of the weights and biases, and then multiplies this by the learning rate to get the change in weights and biases for the next epoch. A large gradient may, however, cause the weights and biases to change by a large amount, and thus miss a set of optimal values. Gradient clipping enables one to set a maximum gradient, such that the weights and biases do not change too drastically between epochs.

Results

I tried each of the above optimisations individually to see which would cause an improvement in performance. I found that the early stopping, a learning rate scheduler, normalisation, and attempting different architectures all caused an improvement in performance. Putting these optimisations together gave an accuracy 80.50% ± 2.05, giving me the best model thus far. Having reached a model with accuracy exceeding 80%, I wished to take a look at most relevant features in the dataset, to see which features were most important in distinguishing H. armigera and H. zea moths.

Relevant Features

To determine the most relevant features, I went back to my best LDA model (not including hybrids and grouping dorsal/ventral data points together), as the neural network's complexity made it difficult to determine the individual impact of each feature. I also artificially balanced the dataset to set a baseline (as described in further detail later). I removed all input features from the dataset, and reintroduced them one at a time, training and testing the model each time. After running the model multiple times and averaging the results, I found the 15 most relevant features as follows:

1. colour_1_g_v: 64.73%
2. colour_1_r_v: 62.97%
3. colour_1_b_v: 61.83%
4. colour_0_g_v: 58.19%
5. colour_0_b_v: 57.90%
6. minferet_v: 57.00%
7. minor_v: 56.98%
8. minferet_d: 56.53%
9. colour_0_r_v: 56.37%
10. minor_d: 55.85%
11. area_v: 55.71%
12. area_d: 55.40%
13. colour_1_g_d: 55.18%
14. colour_0_r_d: 55.18%
15. colour_1_b_d: 54.65%

The suffixes _v and _d refer to the ventral and dorsal sides respectively, and the percentages represent the accuracy from a baseline of 50%. This suggested that ventral colour was by far the most important feature, although different patterns emerged when running the script for each wing individually.

Whilst there were unsurprisingly no differences between left and right wings, I found area, Feret diameter, and minor axis length to be most important for the front wings, with ventral colour being the most important feature at the rear. This was interesting and gave some insight which could be used to create more specific models for each wing location if I had the time in this project to do so.

Another Problem: Class Imbalance

In analysing all this data, I stumbled across another issue: class imbalances. Over 72% of all the dataset was made up of H. armigera moths, with the rest being split evenly between H. zea and hybrids. This made the accuracy values I had achieved till now a lot less impressive. There are two potential baselines when evaluating a model: the zero-weight baseline – the accuracy of a model which always predicts the majority class (i.e., 72%); and the random-rate baseline – the accuracy of a model that guesses randomly, but in proportion to the class balances (approximately 56.4% in this case). Whilst my best-performing model beat both baselines, it was clear that the imbalance was having a significant impact on my achieved accuracy values, and so the best measure was to artificially balance the dataset.

Balancing the dataset

The dataset could be balanced through two methods: under-sampling, where the plurality class is reduced randomly until the dataset is balanced; or over-sampling, where samples from the minority class are duplicated/interpolated until balanced. Both seemed to give similar results when attempted on the extracted features neural network, giving an accuracy of 65.31% ± 6.91, far beyond the theoretical zero-weight/random-rate baselines of 33.33% and 32.67% respectively. This suggested that the gathered results for the neural network till now were not solely due to the class imbalance, and that the model was indeed learning from the data. It also showcased how accuracy wasn't the best metric of the model's performance, and thus I switched to using precision and recall scores instead.

Confusion Matrices and Precision/Recall

Confusion matrices visualise a model's performance as a table of predicted classes against actual classes, with the values in each cell represent the number of samples that fall into said category. Running the extracted features neural network many times over gave the following confusion matrices:

Original		Predicted		
		Armigera	Hybrid	Zea
True	Armigera	*15561*	119	918
	Hybrid	1929	*585*	460
	Zea	1462	99	*1470*

Balanced		Predicted		
		Armigera	Hybrid	Zea
True	Armigera	*9638*	4312	2709
	Hybrid	3850	*8925*	3884
	Zea	1416	2556	*12687*

Clearly, the original model was incredibly accurate for H. armigera moths, but incredibly poor with hybrid and H. zea species. The balanced model, however, performed well across all classes, especially with zea moths. This can be better quantified with precision and recall scores, shown below:

Original	Precision	Recall
Armigera	0.821	0.938
Hybrid	0.729	0.197
Zea	0.516	0.485

Balanced	Precision	Recall
Armigera	0.647	0.579
Hybrid	0.565	0.536
Zea	0.658	0.762

The precision score represents the accuracy of the model when predicting a given class, and the recall score represents the proportion of samples in each class that the model correctly predicts. The poor

performance of the original model on hybrid and zea moths is clearly shown here, with it only correctly identifying 19.7% of hybrid moths and 48.5% of zea moths. The balanced model, however, achieved much better precision and recall scores across all 3 categories.

This showed how the balance across the dataset could be manually manipulated to improve performance for specific use-cases. In this case, however, the main aim was to identify armigera moths (i.e., the 'dangerous' species), and so the high precision and recall values for armigera moths in the original model meant that I was confident that this model would remain the best for the purposes of this project.

Conclusion

In conclusion, I had most certainly achieved my initial goal of creating a machine learning model to distinguish between H. armigera and H. zea moths. Whilst by no means perfect, it definitely accomplished its intended purpose to act as an aid to farmers in the field and could absolutely be improved upon with even more data or further refinement. That being said, there are already many potential further changes I could make if I had more time on this project.

Potential Further Steps

The first change I could make would be to use the understanding I gathered about the most relevant features for each wing to help develop a more refined model – such a model would treat each wing separately, and would be able to distinguish between the two species based on the most relevant features for each wing, rather than generalising across all wings, and this would likely allow for a solid increase in performance. Additionally, I could make use of the continuous admixture data to create a regression model rather than a classification model. The models I have made in this project have no understanding of the links between armigera, zea, and hybrid (specifically that a hybrid is a mix of the two), and so a regression model could make use of this to potentially improve accuracy. Whilst I did start experimenting with this a little bit over the course of the project, I did not have enough time to fully explore this avenue, and so it would be interesting to see how well such a model could perform.