National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Preena Darshini
x22238590

School of Computing
National College of Ireland

Supervisor: Prof. Barry Haycock

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	……………….......Preena Darshini . ……………………………………..…………………
Student ID:	………………………x22238590……………..……………………………………………..……
Programme:	…………….MSc Data Analytics……………… **Year:** ……2024……..…..
Module:	……………………MSc Research Project ……………………………………………….………
Lecturer:	……………………Professor Barry Haycock…………………………………….………
Submission Due Date:	…………………………12/08/2024…….……………………………………………….………
Project Title:	Assessing the Efficacy of EfficientNet, Inception, and ResNet for Wildlife Species Identification ………………………………………………………………………………………….………
Word Count:	2196 ……………………………………… **Page Count:** ……………..35……………..……..………

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

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

Signature: darshini ……………………………………………………………………………………………

Date: 10/08/2024 ……………………………………………………………………………………………

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

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

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

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

Configuration Manual

Preena Darshini
Student ID: x22238590

There are two parts to this Configuration Manual. Part 1 for the dataset which was later not used for certain reasons which will be discussed. Part 2 for the dataset that was used for developing the current ICT solution for this MSc in Research Project.

Part 1: Initial Dataset (Not used in Final Solution)

1 Section 1: Acquiring the Dataset and Downloading

- The dataset was downloaded from the Google Cloud Storage folder gs://public-datasets-lila/Caltech-unzipped/cct_images (105 GB) along with the metadata (44 MB).
- Initially, there was a shortage of space on the system so made use of an external HDD.
- It took approximately 6 hours to download the zip folder and metadata. It took almost 3 hours to extract the data and metadata from the zip folders.
- This was followed by downloading Python through Windows PowerShell. Ran the ‘pip’ command and upgraded to the latest version.
- Navigated to the external HDD using ‘cd’ command and opened a new Jupyter notebook named “Camtrap.ipynb”.
- Imported necessary modules and paths to the metadata and image dataset were defined.
- The metadata file which is in JSON format was read and loaded into the Python directory.
- Summary of the metadata was displayed with 245,118 images with 3 columns namely id, category_id and image_id.
- Even tried breaking the file stored in the external HDD into 1GB zip files using 7-Zip for faster uploading of images.
- But it took very long to process the large number of images. Metadata file loaded faster due to its small size.
- Ran out of disk space even on Colab after repeating the same steps as above so purchased Colab Pro with additional compute units.
- Decided to access images directly from GCS based on the link provided.
- Lost progress multiple times due to issues like system sleep, Wi-Fi disconnected, and system shut down.
- To overcome this, checkpoint.json was created to continue progress from the last checkpoint.

2 Section 2: Data storage and preprocessing on Colab Pro

- Necessary Libraries were imported as shown in Figure 1.

```
import os
import pandas as pd
from PIL import Image
import io
from tqdm import tqdm
import json
```

```
import tensorflow as tf
import efficientnet.tfkeras as efn
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Figure 1: Importing necessary libraries

- Mounted to Google Drive to save checkpoints as shown in Figure 2.

```
# Mounting to Google Drive
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
# Defining the path to save the checkpoint in my Google Drive
checkpoint_file = '/content/drive/MyDrive/checkpoint.json'
```

Figure 2: Mounting to drive and checkpoint_file definition

3 Section 3: Processing Images and Extracting Metadata

- A function "list_blobs" to list all the blobs in the GCS bucket Figure 3.

```
def list_blobs(bucket, prefix):
#Listing all the blobs in the bucket.
    return list(bucket.list_blobs(prefix=prefix))
```

Figure 3: Function to list blobs

- A function "process_images_from_gcs" to download and process images batch-wise while creating a checkpoint to resume from the last point of interruption as seen in Figure 4.

```
def process_images_from_gcs(blobs, batch_size=100, checkpoint_file='/content/drive/MyDrive/checkpoint.json'):
#Downloading and process images in batches.
    processed_data = []
    batch = []
    start_index = 0

    # Loading checkpoint if exists
    if os.path.exists(checkpoint_file):
        with open(checkpoint_file, 'r') as f:
            checkpoint = json.load(f)
            processed_data = checkpoint['processed_data']
            start_index = checkpoint['index']

    total_blobs = len(blobs)
    with tqdm(total=total_blobs, initial=start_index, desc="Processing images") as pbar:
        for i, blob in enumerate(blobs):
            if i < start_index:
                continue
            batch.append(blob)
            if len(batch) >= batch_size:
                process_batch(batch, processed_data)
                pbar.update(len(batch))
                batch = []

                # Save checkpoint
                with open(checkpoint_file, 'w') as f:
                    json.dump({'processed_data': processed_data, 'index': i}, f)

        # Processing the remaining images in the last batch
        if batch:
            process_batch(batch, processed_data)
            pbar.update(len(batch))
        # Processing the remaining images in the last batch
        if batch:
            process_batch(batch, processed_data)
            pbar.update(len(batch))

            # Save final checkpoint
            with open(checkpoint_file, 'w') as f:
                json.dump({'processed_data': processed_data, 'index': i+1}, f)

    return pd.DataFrame(processed_data)
```

Figure 4: Function to process images from GCS

- A function "process_batch" to process each batch of images and extract the metadata as shown in Figure 5.

```python
def process_batch(batch, processed_data):
    for blob in batch:
        # Read the image from GCS
        image_data = blob.download_as_bytes()
        image = Image.open(io.BytesIO(image_data))
        processed_data.append({
            'filename': blob.name,
            'width': image.width,
            'height': image.height,
        })

# Listing all images in the GCS bucket
blobs = list_blobs(bucket, 'caltech-unzipped/cct_images')

# Processing images in batches and collecting metadata
metadata_df = process_images_from_gcs(blobs, checkpoint_file=checkpoint_file)

# Saving the metadata to a CSV file
metadata_df.to_csv('/content/metadata.csv', index=False)
print("Metadata processing completed.")
```

Figure 5: Processing images batch-wise and displaying metadata

4 Section 4: Displaying Statistics and Sample Images

- A function "display_first_images" is used to display the first few images from the GCS bucket as seen in Figure 6.

```python
# Displaying the first few images
import matplotlib.pyplot as plt

def display_first_images(blobs, n=5):
    #Displaying the first 5 images.
    count = 0
    for blob in blobs:
        if count >= n:
            break
        # Read the image from the GCS
        image_data = blob.download_as_bytes()
        image = Image.open(io.BytesIO(image_data))
        plt.imshow(image)
        plt.axis('off')
        plt.show()
        count += 1


display_first_images(blobs, n=5)
```

Figure 6: Displaying head of dataset

- Basic summary statistics of the metadata are displayed as shown in Figure 7.

```python
# Displaying basic statistics of metadata
print(metadata_df.describe())
```

Figure 7: Summary Statistics

5 Section 5: Model Training

- Class named “GCSImageDataGenerator” for loading the images in batches from GCS as shown in Figure 8 below.

```
class GCSImageDataGenerator(tf.keras.utils.Sequence):
    def __init__(self, blobs, batch_size, target_size, label_to_index, subset=None):
        self.blobs = [blob for blob in blobs]
        self.batch_size = batch_size
        self.target_size = target_size
        self.label_to_index = label_to_index
        self.subset = subset
        self.indices = list(range(len(self.blobs)))
        self.on_epoch_end()

    def __len__(self):
        return int(np.floor(len(self.blobs) / self.batch_size))

    def __getitem__(self, index):
        indices = self.indices[index*self.batch_size:(index+1)*self.batch_size]
        batch_blobs = [self.blobs[i] for i in indices]
        return self.__data_generation(batch_blobs)

    def on_epoch_end(self):
        if self.subset == 'training':
            np.random.shuffle(self.indices)

    def __data_generation(self, batch_blobs):
        X = np.empty((self.batch_size, *self.target_size, 3))
        y = np.empty((self.batch_size), dtype=int)

        for i, blob in enumerate(batch_blobs):
            image_data = blob.download_as_bytes()
            image = Image.open(io.BytesIO(image_data)).resize(self.target_size)
            X[i,] = np.array(image) / 255.0
            y[i] = self.get_label_from_filename(blob.name)

        return X, y

    def get_label_from_filename(self, filename):
        label = os.path.basename(os.path.dirname(filename))
        return self.label_to_index[label]
```

Figure 8: Load images in batches

- Parameters for batch size and the target image size are defined as 32 and (244,244) respectively. The resizing is done to 244 pixels in height and 244 pixels in width. The model processes 32 images each time before the backpropagation step.

```
batch_size = 32
target_size = (224, 224)
```

Figure 9: Batch size and target size set

- Labels are extracted and mapped to the numeric classes as displayed in Figure 10.

```python
# Creating a mapping of labels to numeric classes
def extracting_labels_and_create_mapping(blobs):
    labels = set()
    for blob in blobs:
        label = os.path.basename(os.path.dirname(blob.name))
        labels.add(label)
    label_to_index = {label: index for index, label in enumerate(sorted(labels))}
    return label_to_index

label_to_index = extracting_labels_and_create_mapping(blobs)
```

Figure 10: Labels are mapped

- Data is split into train and test sets as shown below.

```python
# Splitting data into training and test
train_blobs = [blob for i, blob in enumerate(blobs) if i % 5 != 0]  # 80% for training
val_blobs = [blob for i, blob in enumerate(blobs) if i % 5 == 0]    # 20% for testing

train_generator = GCSImageDataGenerator(train_blobs, batch_size, target_size, subset='training')
val_generator = GCSImageDataGenerator(val_blobs, batch_size, target_size, subset='validation')
```

Figure 11: Data splitting

- EfficientNet is defined and used for training as seen below.

```python
# EfficientNet model
model = tf.keras.Sequential([
    efn.EfficientNetB0(input_shape=(224, 224, 3), weights='imagenet', include_top=False),
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(len(set(train_generator.labels)), activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Figure 12: EfficientNet Model

- Model training is done.

```python
# Train the model
history = model.fit(
    train_generator,
    validation_data=val_generator,
    epochs=10
)
```

Figure 13: Training EfficientNet

- Kernal restart issue on Colab while trying to run EfficientNet as shown below.

Level	Message
WARNING	WARNING:root:kernel 5259a48e-43fb-4d01-8ec5-caf39b372edc restarted
WARNING	WARNING:root:kernel 5259a48e-43fb-4d01-8ec5-caf39b372edc restarted

Figure 14: Kernel Restart

- On further investigation it seemed like there was also a mismatch of metadata and image files. This made it difficult to link each metadata with the corresponding image. After considering these issues, it was decided along with the approval of the guide that an alternative dataset would be used.

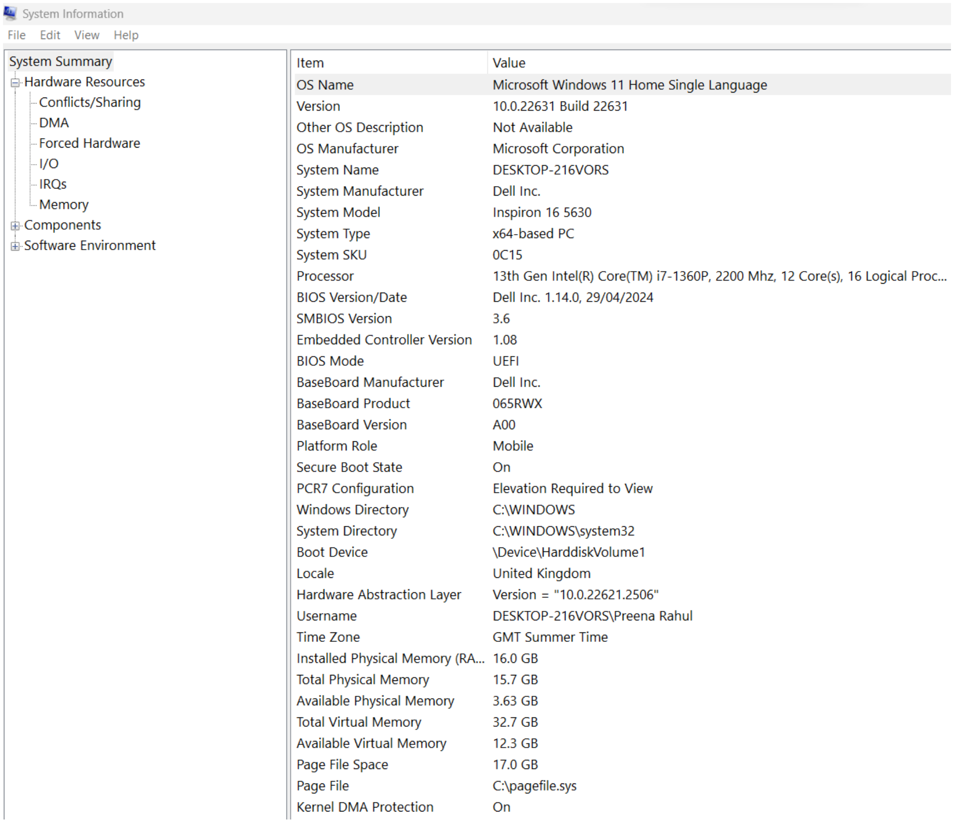
Part 2: Dataset (Used in Final Solution)

1 Section 1: Acquiring the Dataset and Downloading

- The iWildCam 2019 dataset was around 46.68 GB in size. The training set contained 196,086 images from 138 locations in the southern part of California. The test set consisted of 153,730 images from 100 different locations in Idaho.
- The dataset was downloaded on the system which took approximately 2 hours and the test and train zip folders were extracted.

Section 1.1: Hardware Requirements

- The hardware requirements are as shown below in Figure 1.



System Information
File Edit View Help

System Summary
Hardware Resources
Conflicts/Sharing
DMA
Forced Hardware
I/O
IRQs
Memory
Components
Software Environment

Item	Value
OS Name	Microsoft Windows 11 Home Single Language
Version	10.0.22631 Build 22631
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	DESKTOP-216VORS
System Manufacturer	Dell Inc.
System Model	Inspiron 16 5630
System Type	x64-based PC
System SKU	0C15
Processor	13th Gen Intel(R) Core(TM) i7-1360P, 2200 Mhz, 12 Core(s), 16 Logical Proc...
BIOS Version/Date	Dell Inc. 1.14.0, 29/04/2024
SMBIOS Version	3.6
Embedded Controller Version	1.08
BIOS Mode	UEFI
BaseBoard Manufacturer	Dell Inc.
BaseBoard Product	065RWX
BaseBoard Version	A00
Platform Role	Mobile
Secure Boot State	On
PCR7 Configuration	Elevation Required to View
Windows Directory	C:\WINDOWS
System Directory	C:\WINDOWS\system32
Boot Device	\Device\HarddiskVolume1
Locale	United Kingdom
Hardware Abstraction Layer	Version = "10.0.22621.2506"
Username	DESKTOP-216VORS\Preena Rahul
Time Zone	GMT Summer Time
Installed Physical Memory (RA...	16.0 GB
Total Physical Memory	15.7 GB
Available Physical Memory	3.63 GB
Total Virtual Memory	32.7 GB
Available Virtual Memory	12.3 GB
Page File Space	17.0 GB
Page File	C:\pagefile.sys
Kernel DMA Protection	On

Figure 1: Hardware Requirements

Section 1.2: Software Requirements

- Windows 11
- Anaconda Jupyter Notebook

2 Section 2: Importing Necessary Libraries

- The required libraries are installed and imported as shown in Figure 2.

```
!pip install tensorflow
!pip install pandas scikit-learn matplotlib Pillow efficientnet tensorflow_hub
```

```
#Importing necessary libraries
import os
import sys
import pandas as pd
import numpy as np
from skimage.io import imread

#For plotting purposes
import matplotlib.pyplot as plt
from PIL import Image, ImageDraw
from sklearn.utils import class_weight
import seaborn as sns

#For managing files
from keras.preprocessing import image
import zipfile
from sklearn.model_selection import train_test_split

#For machine learning
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import tensorflow_hub as hub

#For model evaluation
from sklearn.metrics import precision_recall_curve, auc, f1_score,accuracy_score, precision_score, recall_score
from keras.callbacks import Callback
from sklearn.utils import class_weight
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, auc
from sklearn.preprocessing import label_binarize
import itertools
```

```
#pre-trained model
#efficientNet, Inception and ResNet
from tensorflow.keras import layers, models
from efficientnet.tfkeras import EfficientNetB0
from efficientnet.tfkeras import center_crop_and_resize, preprocess_input
from tensorflow.keras.applications import InceptionV3, ResNet50
from tensorflow.keras.models import load_model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
```

Figure 2: Importing necessary libraries

Library	Use
os	Function to interact with os, paths and directories
sys	Functions to interact with python interpreter
pandas	For data manipulation activities
numpy	For numerical operations involving arrays and matrices
from skimage.io import imread	For reading image files into array
Matplotlib.pyplot	For creating visualisations
from PIL import Image, ImageDraw	For opening and manipulating images and simple 2D graphics
from sklearn.utils import class_weight	For handling imbalanced datasets
seaborn	For data visualisation

from keras.preprocessing import image	For image loading and processing
import zipfile	For handling zip files
from sklearn.model_selection import train_test_split	For splitting dataset into train and validation
tensorflow	For building machine learning models
from tensorflow.keras.preprocessing.image import ImageDataGenerator	For rescaling and processing images batch-wise
tensorflow_hub	For using pre-trained models
from sklearn.metrics import precision_recall_curve, auc, f1_score, accuracy_score, precision_score, recall_score	For using various performance metrics
from keras.callbacks import Callback	To execute the code at different stages while training
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, auc	For using performance metrics
from sklearn.preprocessing import label_binarize	Used for binarizing labels in case of multi-classification
itertools	For efficiently looping
from tensorflow.keras import layers, models	For importing common layers
from efficientnet.tfkeras import EfficientNetB0	A pre-trained deep learning model
from efficientnet.tfkeras import center_crop_and_resize, preprocess_input	For preprocessing images
from tensorflow.keras.applications import InceptionV3, ResNet50	Pre-trained models
from tensorflow.keras.models import load_model	Used for loading the pre-trained Keras model from the saved files
from tensorflow.keras.optimizers import Adam	Optimisation algorithm
from tensorflow.keras.callbacks import EarlyStopping	To monitor the validation loss

Table 1: List of libraries

- The Table 1 above shows the list of libraries used and its purpose.
- The TensorFlow version is checked as shown below.

```
[3]: tf.__version__

[3]: '2.17.0'
```

Figure 3: TensorFlow Version

3 Section 3: Exploring the Data

- The path to the location where the dataset has been downloaded is defined. Then the files in that path are listed as shown in the below Figure 4.

```
# Checking if data is available in that path where data was downloaded
PATH="C:/Users/Preena Rahul/Desktop/iwildcam-2019-fgvc6"
os.listdir(PATH)
```

```
['sample_submission.csv',
 'test.csv',
 'test_images',
 'test_images.zip',
 'train.csv',
 'train_images',
 'train_images.zip']
```

Figure 4: Data Availability

- A Python dictionary is created which maps the class IDs with the animal names as shown in Figure 5.

```
# classes
classes_wild = {0: 'empty', 1: 'deer', 2: 'moose', 3: 'squirrel', 4: 'rodent', 5: 'small_mammal', \
                6: 'elk', 7: 'pronghorn_antelope', 8: 'rabbit', 9: 'bighorn_sheep', 10: 'fox', 11: 'coyote', \
                12: 'black_bear', 13: 'raccoon', 14: 'skunk', 15: 'wolf', 16: 'bobcat', 17: 'cat',\
                18: 'dog', 19: 'opossum', 20: 'bison', 21: 'mountain_goat', 22: 'mountain_lion'}
```

Figure 5: Classes

- The train and test directories are defined for easy access to the images which can be seen in Figure 6.

```
10]: # listing the locations of train and test set
train_images_directory="C:/Users/Preena Rahul/Desktop/iwildcam-2019-fgvc6/train_images"

test_images_directory="C:/Users/Preena Rahul/Desktop/iwildcam-2019-fgvc6/test_images"
```

Figure 6: Train and test directory paths

- The number of train and test images are listed as seen in Figure 7.

```
[12]: # Directory for test images and number ( Similarly for train )
test_image_files = list(os.listdir(test_images_directory))
print("Number of image files: test: {}".format(len(test_image_files)))
```

```
Number of image files: test: 153730
```

```
[14]: train_image_files = list(os.listdir(train_images_directory))
print("Number of image files: train: {}".format(len(train_image_files)))
```

```
Number of image files: train: 196086
```

Figure 7: Number of train and test images

- As shown in Figure 8, CSV files are loaded in DataFrames and displayed.

```
[17]: train_df = pd.read_csv(os.path.join(PATH, 'train.csv'))
test_df = pd.read_csv(os.path.join(PATH, 'test.csv'))

display(train_df.head())
display(test_df.head())
```

	category_id	date_captured	file_name	frame_num	id	location	rights_holder	seq_id	seq_num_frames	width	height
0	19	2011-05-13 23:43:18	5998cfa4-23d2-11e8-a6a3-ec086b02610b.jpg	1	5998cfa4-23d2-11e8-a6a3-ec086b02610b	33	Justin Brown	6f084ccc-5567-11e8-bc84-dca9047ef277	3	1024	747
1	19	2012-03-17 03:48:44	588a679f-23d2-11e8-a6a3-ec086b02610b.jpg	2	588a679f-23d2-11e8-a6a3-ec086b02610b	115	Justin Brown	6f12067d-5567-11e8-b3c0-dca9047ef277	3	1024	747
2	0	2014-05-11 11:56:46	59279ce3-23d2-11e8-a6a3-ec086b02610b.jpg	1	59279ce3-23d2-11e8-a6a3-ec086b02610b	96	Erin Boydston	6faa92d1-5567-11e8-b1ae-dca9047ef277	1	1024	747
3	0	2013-10-06 02:00:00	5a2af4ab-23d2-11e8-a6a3-ec086b02610b.jpg	1	5a2af4ab-23d2-11e8-a6a3-ec086b02610b	57	Erin Boydston	6f7d4702-5567-11e8-9e03-dca9047ef277	1	1024	747
4	0	2011-07-12 13:11:16	599fbd89-23d2-11e8-a6a3-ec086b02610b.jpg	3	599fbd89-23d2-11e8-a6a3-ec086b02610b	46	Justin Brown	6f1728a1-5567-11e8-9be7-dca9047ef277	3	1024	747

	date_captured	file_name	frame_num	id	location	rights_holder	seq_id	seq_num_frames	width	height
0	03-Jan-2016 11:30:56	bce932f6-2bf6-11e9-bcad-06f10d5896c4.jpg	1	bce932f6-2bf6-11e9-bcad-06f10d5896c4	37	Idaho Department of Fish and Game	6e9ac61c-2e32-11e9-90ef-dca9047ef277	5	1024	726
1	03-Jan-2016 11:30:57	bce932f7-2bf6-11e9-bcad-06f10d5896c4.jpg	2	bce932f7-2bf6-11e9-bcad-06f10d5896c4	37	Idaho Department of Fish and Game	6e9ac61c-2e32-11e9-90ef-dca9047ef277	5	1024	726
2	03-Jan-2016 11:30:58	bce932f8-2bf6-11e9-bcad-06f10d5896c4.jpg	3	bce932f8-2bf6-11e9-bcad-06f10d5896c4	37	Idaho Department of Fish and Game	6e9ac61c-2e32-11e9-90ef-dca9047ef277	5	1024	726
3	03-Jan-2016	bce932f9-2bf6-11e9-	4	bce932f9-2bf6-11e9-	37	Idaho Department of	6e9ac61c-2e32-11e9-	5	1024	726

Figure 8: Head of train.csv and test.csv DataFrames

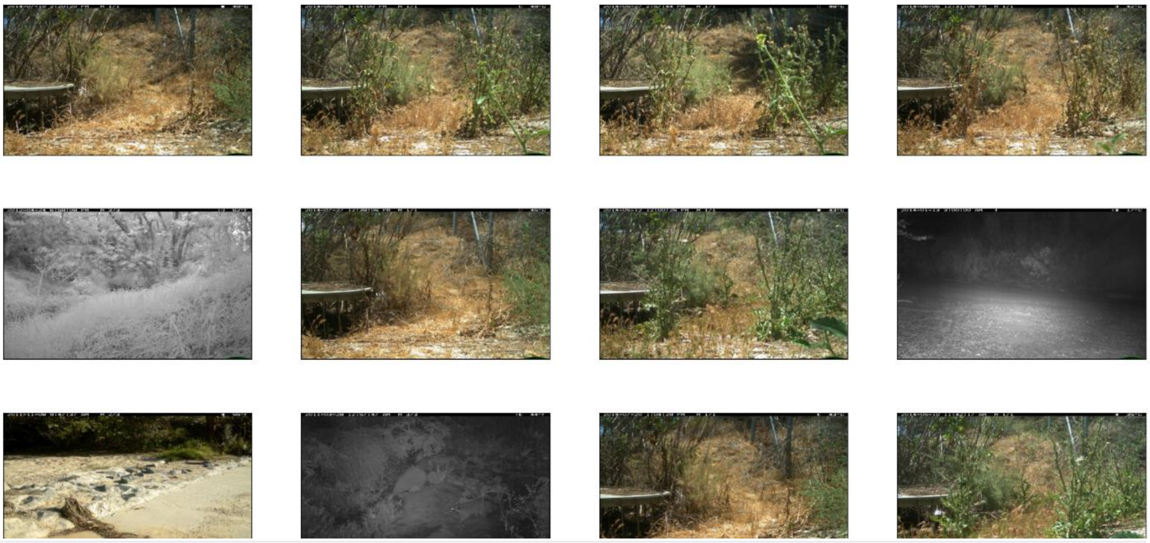
- Figure 9 shows the DataFrames information which contains information like the number of entries, columns and its data types, and memory usage of the DataFrames. Figure 10 shows 16 sample images from the train images directory and test images directory.

```
display(train_df.info())
display(test_df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 196299 entries, 0 to 196298
Data columns (total 11 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   category_id     196299 non-null  int64
 1   date_captured   196299 non-null  object
 2   file_name       196299 non-null  object
 3   frame_num       196299 non-null  int64
 4   id              196299 non-null  object
 5   location        196299 non-null  int64
 6   rights_holder   196299 non-null  object
 7   seq_id          196299 non-null  object
 8   seq_num_frames  196299 non-null  int64
 9   width           196299 non-null  int64
 10  height          196299 non-null  int64
dtypes: int64(6), object(5)
memory usage: 16.5+ MB
None
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 153730 entries, 0 to 153729
Data columns (total 10 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   date_captured   153730 non-null  object
 1   file_name       153730 non-null  object
 2   frame_num       153730 non-null  int64
 3   id              153730 non-null  object
 4   location        153730 non-null  int64
 5   rights_holder   153730 non-null  object
 6   seq_id          153730 non-null  object
 7   seq_num_frames  153730 non-null  int64
 8   width           153730 non-null  int64
 9   height          153730 non-null  int64
dtypes: int64(5), object(5)
memory usage: 11.7+ MB
None
```

Figure 9: Information of DataFrames

```python
# Displaying sample train images
fig = plt.figure(figsize=(25, 16))
for i, im_path in enumerate(train_image_files[:16]):
    ax = fig.add_subplot(4, 4, i+1, xticks=[], yticks=[])
    im = Image.open(os.path.join(train_images_directory, im_path))
    im = im.resize((480, 270))
    plt.imshow(im)
plt.show()
```



```python
# Displaying sample test images
fig = plt.figure(figsize=(25, 16))
for i, im_path in enumerate(test_image_files[:16]):
    ax = fig.add_subplot(4, 4, i+1, xticks=[], yticks=[])
    im = Image.open(os.path.join(test_images_directory, im_path))
    im = im.resize((480, 270))
    plt.imshow(im)
plt.show()
```

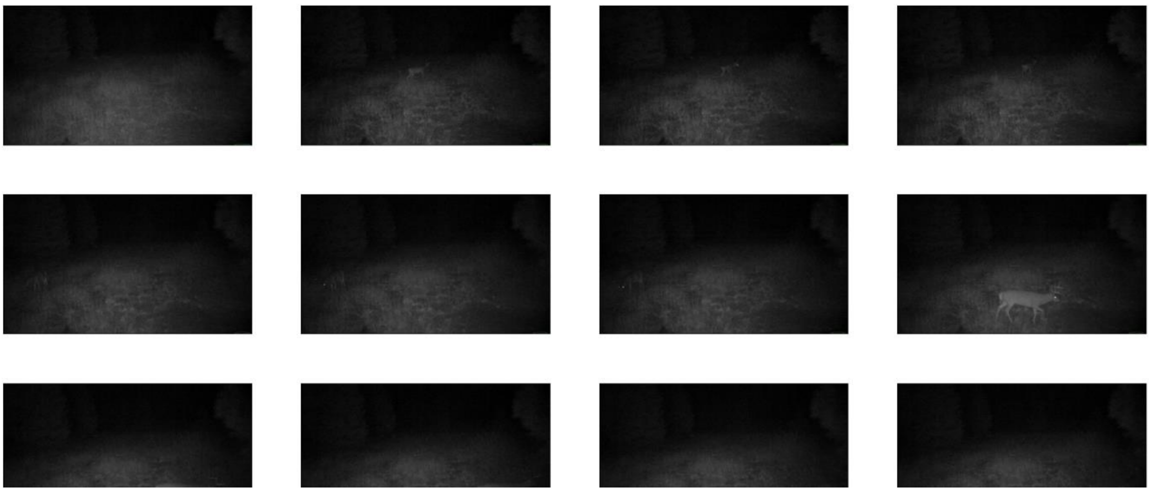


Figure 10: Sample images from train and test directories

4 Section 4: Manipulating the Data

- The class names are then mapped to the category IDs and after appending, the first few rows of the train_df are viewed as seen in the below as shown in Figure 11.

```
train_df['classes_wild'] = train_df['category_id'].apply(lambda cw: classes_wild[cw])
```

```
train_df.head()
```

	category_id	date_captured	file_name	frame_num	id	location	rights_holder	seq_id	seq_num_frames	width	height	classes_wild
0	19	2011-05-13 23:43:18	5998cfa4-23d2-11e8-a6a3-ec086b02610b.jpg	1	5998cfa4-23d2-11e8-a6a3-ec086b02610b	33	Justin Brown	6f084ccc-5567-11e8-bc84-dca9047ef277	3	1024	747	opossum
1	19	2012-03-17 03:48:44	588a679f-23d2-11e8-a6a3-ec086b02610b.jpg	2	588a679f-23d2-11e8-a6a3-ec086b02610b	115	Justin Brown	6f12067d-5567-11e8-b3c0-dca9047ef277	3	1024	747	opossum
2	0	2014-05-11 11:56:46	59279ce3-23d2-11e8-a6a3-ec086b02610b.jpg	1	59279ce3-23d2-11e8-a6a3-ec086b02610b	96	Erin Boydston	6faa92d1-5567-11e8-b1ae-dca9047ef277	1	1024	747	empty
3	0	2013-10-06 02:00:00	5a2af4ab-23d2-11e8-a6a3-ec086b02610b.jpg	1	5a2af4ab-23d2-11e8-a6a3-ec086b02610b	57	Erin Boydston	6f7d4702-5567-11e8-9e03-dca9047ef277	1	1024	747	empty
4	0	2011-07-12 13:11:16	599fbd89-23d2-11e8-a6a3-ec086b02610b.jpg	3	599fbd89-23d2-11e8-a6a3-ec086b02610b	46	Justin Brown	6f1728a1-5567-11e8-9be7-dca9047ef277	3	1024	747	empty

Figure 11: Mapping class names to IDs

- The column names of both train and test DataFrames are viewed in Figure 12.

```
print("Columns in train_df:", train_df.columns)
print("Columns in test_df:", test_df.columns)
```

```
Columns in train_df: Index(['category_id', 'date_captured', 'file_name', 'frame_num', 'id',
       'location', 'rights_holder', 'seq_id', 'seq_num_frames', 'width',
       'height', 'classes_wild'],
      dtype='object')
Columns in test_df: Index(['date_captured', 'file_name', 'frame_num', 'id', 'location',
       'rights_holder', 'seq_id', 'seq_num_frames', 'width', 'height'],
      dtype='object')
```

Figure 12: Columns in train and test

- As shown in Figure 13, the file name column in train_df and test _df are updated to include the path of each image file.

```
# Include complete path of each image file
train_df['file_name'] = train_df['file_name'].apply(lambda x: os.path.join(train_images_directory, os.path.basename(x)))
test_df['file_name'] = test_df['file_name'].apply(lambda x: os.path.join(test_images_directory, os.path.basename(x)))
```

Figure 13: Including the complete path of each file

- The class distribution is plotted as seen below in Figure 14.

```python
# Function to plot class distribution
def plot_class_distribution(y, title):
    plt.figure(figsize=(12, 6))
    sns.countplot(x=y)
    plt.title(title)
    plt.xlabel('Class')
    plt.ylabel('Frequency')
    plt.show()

plot_class_distribution(train_df['category_id'], 'Original Class Distribution')
```

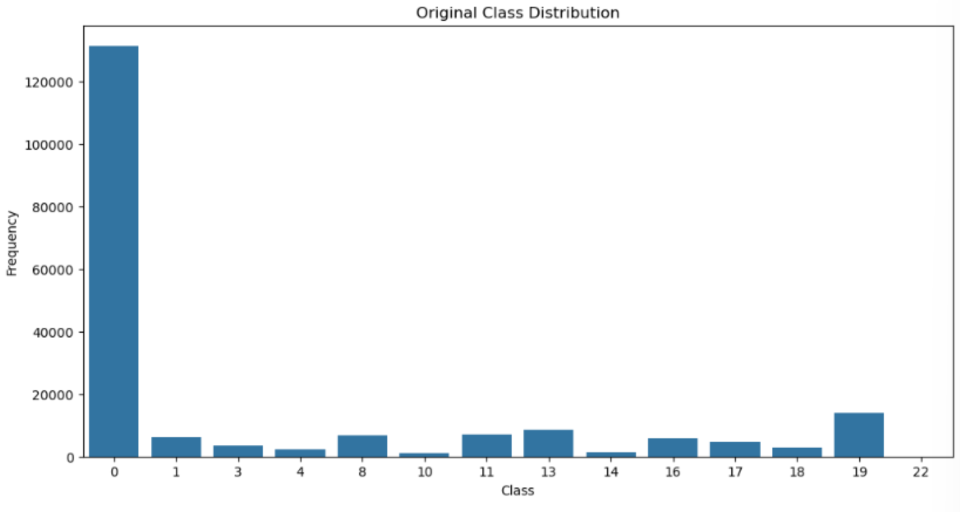


Figure 14: Class distribution function and plot

- Figure 15 shows the splitting of the train dataset into train and validation sets. The train set contains 157,039 images and the validation set contains 39,260 images each with 12 columns. The class weights are also computed and the dictionary of class weights is displayed.

```python
# Splitting dataset into train and validation
x_train, x_val = train_test_split(train_df, test_size=0.2, random_state=42)
print(x_train.shape, x_val.shape)
```

```
(157039, 12) (39260, 12)
```

```python
# Computing class weights
class_weights = class_weight.compute_class_weight(
    class_weight='balanced',
    classes=np.unique(train_df['category_id']),
    y=train_df['category_id']
)

class_weights_dict = dict(enumerate(class_weights))
print(class_weights_dict)
```

```
{0: 0.1066611678560833, 1: 2.2978297513695742, 2: 4.126355839569495, 3: 6.344505494505494,
4: 2.0209508709796977, 5: 12.82832309502026, 6: 1.9449794899233102, 7: 1.6260416494093868,
8: 10.302246247507085, 9: 2.346670651524208, 10: 2.9462822321616184, 11: 4.619887032242881,
12: 0.9939995138846691, 13: 424.88961038961037}
```

Figure 15: Train-val split and class weights computation

- Figure 16 shows the class distribution after applying class weights.

```
# Creating a DataFrame to visualize class weights
weighted_counts = pd.DataFrame({
    'class': np.unique(train_df['category_id']),
    'original_count': train_df['category_id'].value_counts().sort_index().values,
    'weighted_count': train_df['category_id'].value_counts().sort_index().values * class_weights
})

# Plotting the weighted class distribution
plt.figure(figsize=(12, 6))
bar_width = 0.4
index = np.arange(len(np.unique(train_df['category_id'])))
plt.bar(index, weighted_counts['original_count'], bar_width, label='Original Count')
plt.bar(index + bar_width, weighted_counts['weighted_count'], bar_width, label='Weighted Count')
plt.xlabel('Class')
plt.ylabel('Frequency')
plt.title('Class Distribution Before and After Applying Class Weights')
plt.xticks(index + bar_width / 2, weighted_counts['class'])
plt.legend()
plt.show()
```

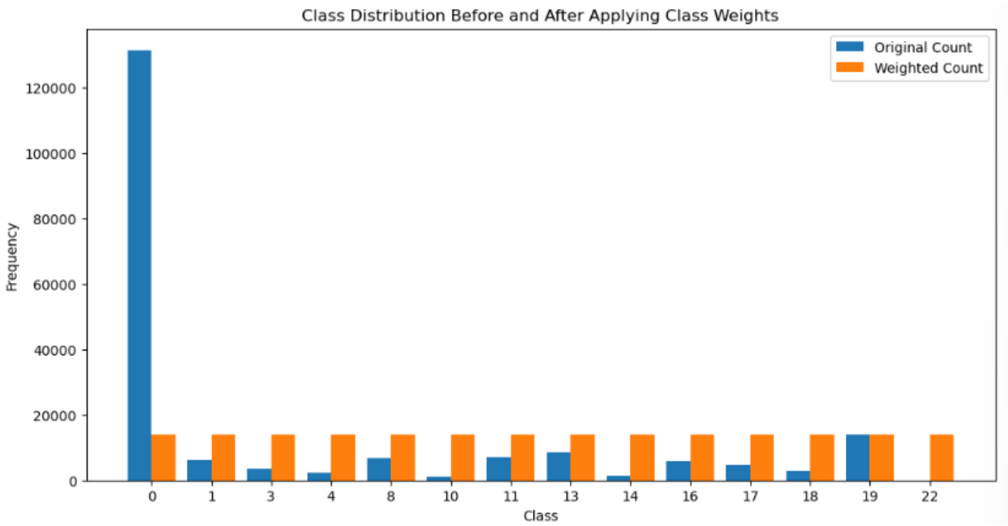


Figure 16: Original and weighted class distribution

- Figure 17 shows that rescaling is done to change pixel values from [0,255] to [0,1] for better and quicker training. The train data generator also has a split of 25% for validation.
- The train and validation generators are set up with the necessary parameters as seen in Figure 17.

```
[51]: test_datagen = ImageDataGenerator(rescale = 1./255)

      train_datagen=ImageDataGenerator(rescale=1./255,
                                       validation_split=0.25
                                      )
```

```
[53]: train_generator = train_datagen.flow_from_dataframe(
                        dataframe=x_train,
                        directory=train_images_directory,
                        x_col="file_name",
                        y_col="classes_wild",
                        subset="training",
                        batch_size=64,
                        seed=424,
                        shuffle=True,
                        class_mode="categorical",
                        target_size=(128, 128))

      valid_generator = train_datagen.flow_from_dataframe(
                        dataframe=x_train,
                        directory=train_images_directory,
                        x_col="file_name",
                        y_col="classes_wild",
                        subset="validation",
                        batch_size=64,
                        seed=424,
                        shuffle=True,
                        class_mode="categorical",
                        target_size=(128, 128))

      Found 117780 validated image filenames belonging to 14 classes.
      Found 39259 validated image filenames belonging to 14 classes.
```

Figure 17: Rescaling and data generators

- The class indices are also printed in Figure 18.

```
print(train_generator.class_indices)

{'bobcat': 0, 'cat': 1, 'coyote': 2, 'deer': 3, 'dog': 4, 'empty': 5, 'fox': 6, 'mountain_l
ion': 7, 'opossum': 8, 'rabbit': 9, 'raccoon': 10, 'rodent': 11, 'skunk': 12, 'squirrel': 1
3}

print(valid_generator.class_indices)

{'bobcat': 0, 'cat': 1, 'coyote': 2, 'deer': 3, 'dog': 4, 'empty': 5, 'fox': 6, 'mountain_l
ion': 7, 'opossum': 8, 'rabbit': 9, 'raccoon': 10, 'rodent': 11, 'skunk': 12, 'squirrel': 1
3}
```

Figure 18: Class indices

5 Section 5: Modeling

- The number of classes in the train DataFrame is checked, the EfficientNet model is built, and the model summary is displayed as shown in Figure 19.

```
num_classes = train_df['classes_wild'].nunique()
print(f"Number of classes: {num_classes}")
```

```
Number of classes: 14
```

4.1. EfficientNet

```
# Build EfficientNet Model
efficientnet_model = EfficientNetB0(weights='imagenet', include_top=False, input_shape=(128, 128, 3))
efficientnet_model.trainable = False

inputs = tf.keras.Input(shape=(128, 128, 3))
x = efficientnet_model(inputs, training=False)
x = layers.GlobalAveragePooling2D()(x)
outputs = layers.Dense(num_classes, activation='softmax')(x)
efficientnet_model = models.Model(inputs, outputs)

efficientnet_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
efficientnet_model.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 128, 128, 3)	0
efficientnet-b0 (Functional)	(None, 4, 4, 1280)	4,049,564
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1280)	0
dense (Dense)	(None, 14)	17,934

Total params: 4,067,498 (15.52 MB)
Trainable params: 17,934 (70.05 KB)
Non-trainable params: 4,049,564 (15.45 MB)

Figure 19: Verifying the number of classes and Building EfficientNet

- The EfficientNet model is trained and after each epoch, the accuracy and loss for both training and validation sets are printed as seen in Figure 20.

```
# Training EfficientNet
early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3, verbose=1)
history = efficientnet_model.fit(
    train_generator,
    validation_data=valid_generator,
    steps_per_epoch=100,
    epochs=20,
    batch_size=64,
    validation_steps=50,
    class_weight=class_weights_dict,
    callbacks=[early]
)
```

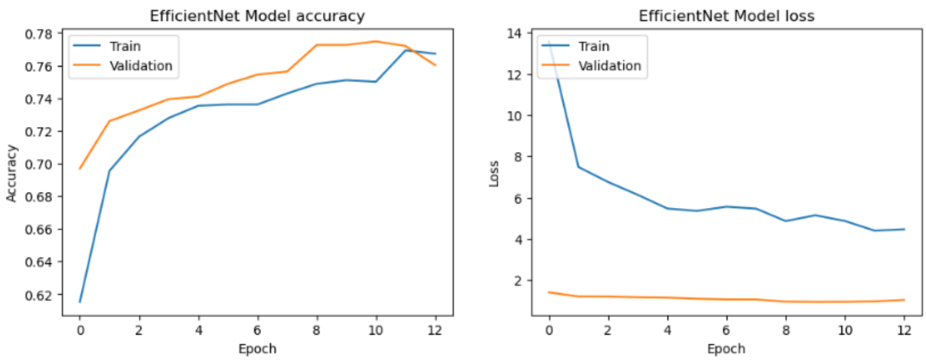
```
Epoch 1/20
C:\Users\Preena Rahul\anaconda3\Lib\site-packages\keras\src\trainers\data_adapters\py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should
call `super().__init__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass these argume
nts to `fit()`, as they will be ignored.
  self._warn_if_super_not_called()
100/100 ━━━━━━━━━━━━━━━━━━━━ 188s 2s/step - accuracy: 0.4945 - loss: 21.8243 - val_accuracy: 0.6969 - val_loss: 1.4025
Epoch 2/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 155s 2s/step - accuracy: 0.6873 - loss: 8.3806 - val_accuracy: 0.7259 - val_loss: 1.2024
Epoch 3/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 145s 1s/step - accuracy: 0.7160 - loss: 6.4711 - val_accuracy: 0.7325 - val_loss: 1.1998
Epoch 4/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 132s 1s/step - accuracy: 0.7310 - loss: 6.0424 - val_accuracy: 0.7394 - val_loss: 1.1634
Epoch 5/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 122s 1s/step - accuracy: 0.7357 - loss: 5.3720 - val_accuracy: 0.7409 - val_loss: 1.1450
Epoch 6/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 90s 910ms/step - accuracy: 0.7305 - loss: 5.0594 - val_accuracy: 0.7487 - val_loss: 1.0929
Epoch 7/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 93s 938ms/step - accuracy: 0.7384 - loss: 5.6726 - val_accuracy: 0.7544 - val_loss: 1.0614
Epoch 8/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 109s 1s/step - accuracy: 0.7383 - loss: 5.6707 - val_accuracy: 0.7563 - val_loss: 1.0576
Epoch 9/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 107s 1s/step - accuracy: 0.7540 - loss: 4.6866 - val_accuracy: 0.7725 - val_loss: 0.9534
Epoch 10/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 109s 1s/step - accuracy: 0.7421 - loss: 6.1157 - val_accuracy: 0.7725 - val_loss: 0.9383
Epoch 11/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 102s 1s/step - accuracy: 0.7505 - loss: 4.7478 - val_accuracy: 0.7747 - val_loss: 0.9433
Epoch 12/20
```

Figure 20: Training EfficientNet Model

- The train and validation accuracy and loss are plotted as shown in the Figure 21 below and evaluation is done on the validation set.

```
# Plotting Training & Validation Accuracy and Loss
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('EfficientNet Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('EfficientNet Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```




```
# Evaluating the model on validation set
val_loss, val_acc = efficientnet_model.evaluate(valid_generator, steps=valid_generator.n // valid_generator.batch_size)
print(f'EfficientNet validation loss: {val_loss} and validation accuracy: {val_acc}')
```

Figure 21: EfficientNet model accuracy and loss

- The presence of test images is checked as shown below in Figure 22.

```
# Checking if test images exist in the test directory
test_images_path = test_images_directory
for filename in test_df['file_name'].head():
    file_path = os.path.join(test_images_path, os.path.basename(filename))
    if not os.path.exists(file_path):
        print(f"File not found: {file_path}")
    else:
        print(f"File exists: {file_path}")
```

```
File exists: C:/Users/Preena Rahul/Desktop/iwildcam-2019-fgvc6/test_images\bce932f6-2bf6-11e9-bcad-06f10d5896c4.jpg
File exists: C:/Users/Preena Rahul/Desktop/iwildcam-2019-fgvc6/test_images\bce932f7-2bf6-11e9-bcad-06f10d5896c4.jpg
File exists: C:/Users/Preena Rahul/Desktop/iwildcam-2019-fgvc6/test_images\bce932f8-2bf6-11e9-bcad-06f10d5896c4.jpg
File exists: C:/Users/Preena Rahul/Desktop/iwildcam-2019-fgvc6/test_images\bce932f9-2bf6-11e9-bcad-06f10d5896c4.jpg
File exists: C:/Users/Preena Rahul/Desktop/iwildcam-2019-fgvc6/test_images\bce932fa-2bf6-11e9-bcad-06f10d5896c4.jpg
```

Figure 22: Checking the presence of test images

- Figure 23 shows predictions are made for the test data and the number of file paths and predictions are verified.

```python
#Predictions for the test set
test_generator = test_datagen.flow_from_dataframe(
    dataframe=test_df,
    directory=test_images_directory,
    x_col="file_name",
    y_col=None,  # No labels in the test set
    batch_size=64,
    seed=424,
    shuffle=False,
    class_mode=None,  # No labels here
    target_size=(128, 128)
)

# Generating predictions
predictions = efficientnet_model.predict(test_generator, steps=test_generator.n // test_generator.batch_size + 1)
predicted_classes = np.argmax(predictions, axis=1)

# Verifying that the file paths are unique
print("Number of filepaths:", len(test_generator.filepaths))
print("Number of predictions:", len(predicted_classes))

# Since there are no true classes, we will not be able to compute accuracy
class_labels = list(train_generator.class_indices.keys())
print(f"Predicted classes: {predicted_classes[:30]}")
```

```
Found 153730 validated image filenames.
C:\Users\Preena Rahul\anaconda3\Lib\site-packages\keras\src\trainers\data_adapters\py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass these arguments to `fit()`, as they will be ignored.
  self._warn_if_super_not_called()
2403/2403 ━━━━━━━━━━━━━━━━━━━━ 3322s 1s/step
Number of filepaths: 153730
Number of predictions: 153730
Predicted classes: [13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13
 13 13 13 13 13 13]
```

Figure 23: Making predictions

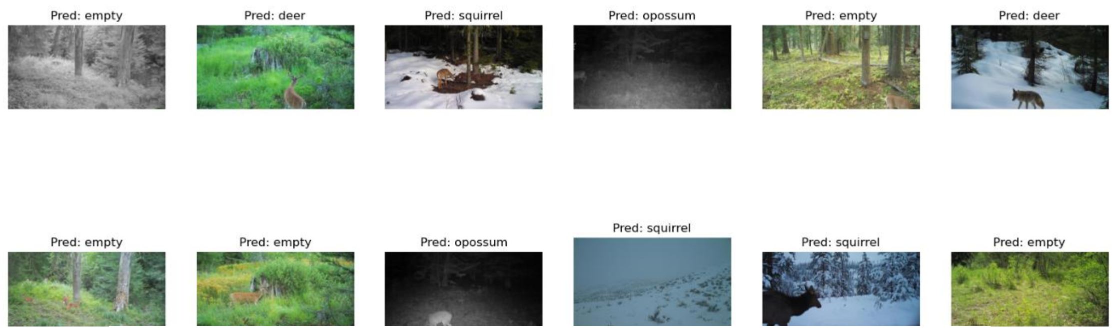
- The test images are displayed with predictions. The model is saved as seen in Figure 24.

```python
import random

# Ensuring that the number of images displayed does not exceed the total number of images in the test set
num_images_to_display = min(30, len(test_generator.filepaths))

# Randomly selecting indices
random_indices = random.sample(range(len(test_generator.filepaths)), num_images_to_display)

# Displaying 30 random images with their predicted labels
plt.figure(figsize=(20, 20))
for i, idx in enumerate(random_indices):
    plt.subplot(5, 6, i + 1)
    img_path = test_generator.filepaths[idx]
    img = plt.imread(img_path)
    plt.imshow(img)
    plt.title(f"Pred: {class_labels[predicted_classes[idx]]}")
    plt.axis('off')
plt.show()
```




```python
efficientnet_model.save('efficientnet_model.keras')
```

Figure 24: Displaying predictions and saving model

- Figure 25 shows the code snippet for building the Inception model. The model is loaded from TensorFlow Keras. The model summary is printed.

```
# InceptionV3 Model

# Building the InceptionV3 Model
inception_model = InceptionV3(weights='imagenet', include_top=False, input_shape=(128, 128, 3))
inception_model.trainable = False

inputs = tf.keras.Input(shape=(128, 128, 3))
x = inception_model(inputs, training=False)
x = layers.GlobalAveragePooling2D()(x)
outputs = layers.Dense(num_classes, activation='softmax')(x)
inception_model = models.Model(inputs, outputs)

inception_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
inception_model.summary()
```

```
Model: "functional_2"
```

Layer (type)	Output Shape	Param #
input_layer_5 (InputLayer)	(None, 128, 128, 3)	0
inception_v3 (Functional)	(None, 2, 2, 2048)	21,802,784
global_average_pooling2d_2 (GlobalAveragePooling2D)	(None, 2048)	0
dense_2 (Dense)	(None, 14)	28,686

```
Total params: 21,831,470 (83.28 MB)
Trainable params: 28,686 (112.05 KB)
Non-trainable params: 21,802,784 (83.17 MB)
```

Figure 25: Building Inception model

- The inception model is then trained and for each epoch, the model's accuracy and loss are displayed as seen in the Figure 26.

```
# Training InceptionV3 Model
early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3, verbose=1)
history = inception_model.fit(
    train_generator,
    validation_data=valid_generator,
    steps_per_epoch=100,
    epochs=20,
    batch_size=64,
    validation_steps=50,
    class_weight=class_weights_dict,  # Added class weights
    callbacks=[early]
)
```

```
Epoch 1/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 179s 2s/step - accuracy: 0.5816 - loss: 18.7419 - val_accuracy: 0.6928 - val_loss: 1.5909
Epoch 2/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 154s 2s/step - accuracy: 0.6981 - loss: 8.9457 - val_accuracy: 0.7244 - val_loss: 1.4066
Epoch 3/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 147s 1s/step - accuracy: 0.7081 - loss: 8.2684 - val_accuracy: 0.7078 - val_loss: 1.3839
Epoch 4/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 144s 1s/step - accuracy: 0.7175 - loss: 7.2262 - val_accuracy: 0.7650 - val_loss: 1.1036
Epoch 5/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 132s 1s/step - accuracy: 0.7357 - loss: 6.5525 - val_accuracy: 0.7225 - val_loss: 1.4873
Epoch 6/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 108s 1s/step - accuracy: 0.7372 - loss: 7.2042 - val_accuracy: 0.7538 - val_loss: 1.1707
Epoch 7/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 109s 1s/step - accuracy: 0.7520 - loss: 5.5905 - val_accuracy: 0.7437 - val_loss: 1.3396
Epoch 7: early stopping
```

Figure 26: Training Inception model

- Figure 27 shows the code snippet where the training and validation accuracy and loss are plotted.

```
# Plotting the training & validation Accuracy and Loss
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('InceptionV3 Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('InceptionV3 Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

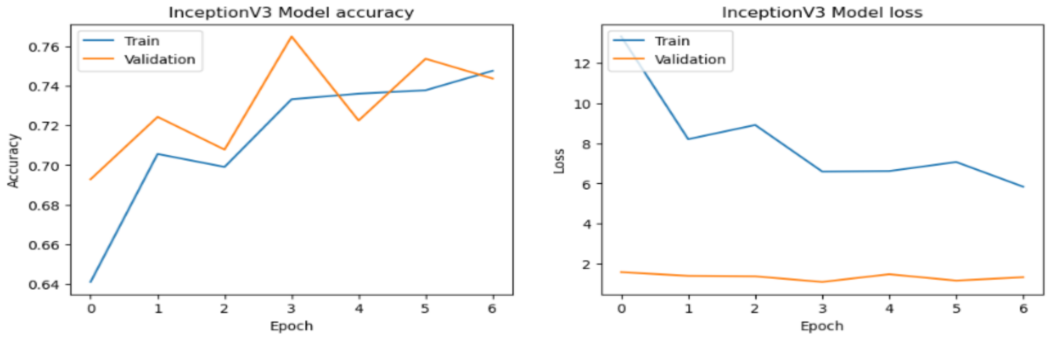


Figure 27: Plots for training and validation accuracy and loss

- The data generator for the test set is generated and predictions are made as shown in Figure 28.

```
# Generating predictions for the test set
test_generator = test_datagen.flow_from_dataframe(
    dataframe=test_df,
    directory=test_images_directory,
    x_col="file_name",
    y_col=None,  # No labels in the test set
    batch_size=64,
    seed=424,
    shuffle=False,
    class_mode=None,  # No labels for this
    target_size=(128, 128)
)
```

```
Found 153730 validated image filenames.
```

```
# Generating predictions
predictions = inception_model.predict(test_generator, steps=test_generator.n // test_generator.batch_size + 1)
predicted_classes = np.argmax(predictions, axis=1)
class_labels = list(train_generator.class_indices.keys())
```

```
C:\Users\Preena Rahul\anaconda3\Lib\site-packages\keras\src\trainers\data_adapters\py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass these arguments to `fit()`, as they will be ignored.
  self._warn_if_super_not_called()
2403/2403 ━━━━━━━━━━━━━━━━━━━━ 3474s 1s/step
```

Figure 28: Test Data Generator and Predictions

- 30 images are randomly selected from the test set and predictions for those images are displayed and the inception model is saved as seen in Figure 29.

```
# Displaying the predictions of 30 random images
random_indices = np.random.choice(len(predicted_classes), 30, replace=False)

plt.figure(figsize=(20, 20))
for i, idx in enumerate(random_indices):
    plt.subplot(5, 6, i + 1)
    img_path = test_generator.filepaths[idx]
    img = plt.imread(img_path)
    plt.imshow(img)
    plt.title(f"Pred: {class_labels[predicted_classes[idx]]}")
    plt.axis('off')
plt.show()
```


```
inception_model.save('inception_model.keras')
```

Figure 29: Displaying images with predictions and saving the model

- The ResNet model is built as shown in Figure 30 and the model summary is displayed.

```
# Building the ResNet50 Model
resnet_model = ResNet50(weights='imagenet', include_top=False, input_shape=(128, 128, 3))
resnet_model.trainable = False

inputs = tf.keras.Input(shape=(128, 128, 3))
x = resnet_model(inputs, training=False)
x = layers.GlobalAveragePooling2D()(x)
outputs = layers.Dense(num_classes, activation='softmax')(x)
resnet_model = models.Model(inputs, outputs)

resnet_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
resnet_model.summary()
```

Model: "functional_3"

Layer (type)	Output Shape	Param #
input_layer_7 (InputLayer)	(None, 128, 128, 3)	0
resnet50 (Functional)	(None, 4, 4, 2048)	23,587,712
global_average_pooling2d_3 (GlobalAveragePooling2D)	(None, 2048)	0
dense_3 (Dense)	(None, 14)	28,686

Total params: 23,616,398 (90.09 MB)
Trainable params: 28,686 (112.05 KB)
Non-trainable params: 23,587,712 (89.98 MB)

Figure 30: Building ResNet model

- Figure 31 shows the code snippet for training the ResNet model. The training and validation accuracy and loss are also displayed for each epoch.

```
# Training the ResNet50 Model
early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3, verbose=1)
history = resnet_model.fit(
    train_generator,
    validation_data=valid_generator,
    steps_per_epoch=100,
    epochs=20,
    batch_size=64,
    validation_steps=50,
    class_weight=class_weights_dict,  # Added the class weights
    callbacks=[early]
)
```

```
Epoch 1/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 257s 2s/step - accuracy: 0.4415 - loss: 29.2507 - val_accurac
y: 0.3047 - val_loss: 2.0673
Epoch 2/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 234s 2s/step - accuracy: 0.4888 - loss: 17.9921 - val_accurac
y: 0.6675 - val_loss: 1.9595
Epoch 3/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 226s 2s/step - accuracy: 0.6492 - loss: 17.1038 - val_accurac
y: 0.6634 - val_loss: 1.8553
Epoch 4/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 222s 2s/step - accuracy: 0.4904 - loss: 18.8067 - val_accurac
y: 0.6406 - val_loss: 1.9158
Epoch 5/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 208s 2s/step - accuracy: 0.6013 - loss: 16.4201 - val_accurac
y: 0.6528 - val_loss: 1.9033
Epoch 6/20
100/100 ━━━━━━━━━━━━━━━━━━━━ 175s 2s/step - accuracy: 0.6402 - loss: 16.2205 - val_accurac
y: 0.5494 - val_loss: 1.9914
Epoch 6: early stopping
```

Figure 31: Training ResNet model

- The model training and validation accuracy and loss are plotted as shown in Figure 32.

```
# Plotting the training & validation Accuracy and Loss
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('ResNet50 Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('ResNet50 Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

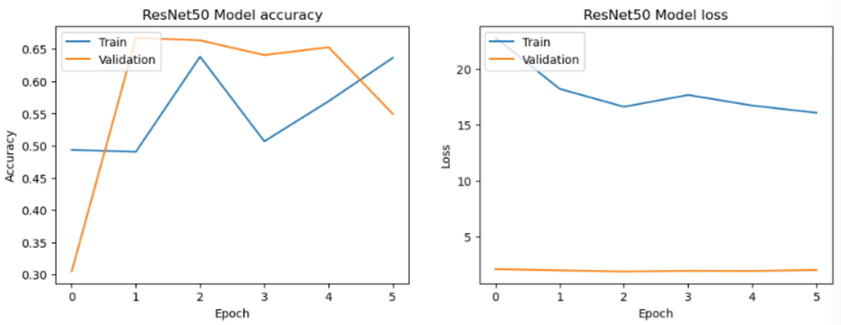


Figure 32: Plot of model training and validation accuracy and loss

- Figure 33 shows the evaluation of the model on test data. The test data generator is created and predictions are made.

```
# Evaluation of the ResNet50 Model
test_generator = test_datagen.flow_from_dataframe(
    dataframe=test_df,
    directory=test_images_directory,
    x_col="file_name",
    y_col=None,  # No labels in the test set
    batch_size=64,
    seed=424,
    shuffle=False,
    class_mode=None,  # No labels here
    target_size=(128, 128)
)
```

```
Found 153730 validated image filenames.
```

```
# Generating predictions
predictions = resnet_model.predict(test_generator, steps=test_generator.n // test_generator.batch_size + 1)
predicted_classes = np.argmax(predictions, axis=1)
class_labels = list(train_generator.class_indices.keys())
```

```
C:\Users\Preena Rahul\anaconda3\Lib\site-packages\keras\src\trainers\data_adapters\py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass these arguments to `fit()`, as they will be ignored.
  self._warn_if_super_not_called()
2403/2403 ━━━━━━━━━━━━━━━━━━━━ 3641s 2s/step
```

Figure 33: Test data generator and predictions of ResNet

- 30 random images are selected to display the predictions from the test set as shown in Figure 34.

```
# Displaying the predictions for 30 random images
random_indices = np.random.choice(len(predicted_classes), 30, replace=False)

plt.figure(figsize=(20, 20))
for i, idx in enumerate(random_indices):
    plt.subplot(5, 6, i + 1)
    img_path = test_generator.filepaths[idx]
    img = plt.imread(img_path)
    plt.imshow(img)
    plt.title(f"Pred: {class_labels[predicted_classes[idx]]}")
    plt.axis('off')
plt.show()
```

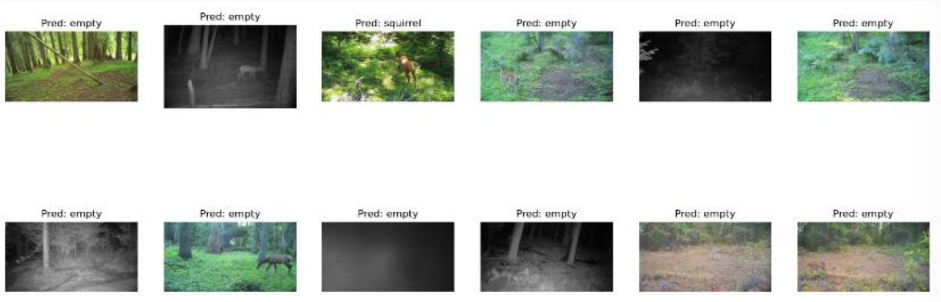


Figure 34: Random images with predictions

- Since the performance of the model was not that great, the ResNet model was fine-tuned as shown in Figure 35.

```
# Defining the learning rate scheduler
def scheduler(epoch, lr):
    if epoch < 10:
        return lr
    else:
        return float(lr * tf.math.exp(-0.1))

lr_schedule = tf.keras.callbacks.LearningRateScheduler(scheduler)

# Building ResNet50 Model again
resnet_model = ResNet50(weights='imagenet', include_top=False, input_shape=(128, 128, 3))
resnet_model.trainable = False

inputs = tf.keras.Input(shape=(128, 128, 3))
x = resnet_model(inputs, training=False)
x = layers.GlobalAveragePooling2D()(x)
outputs = layers.Dense(num_classes, activation='softmax')(x)
resnet_model = models.Model(inputs, outputs)

# Unfreezing the last few layers
for layer in resnet_model.layers[-10:]:
    layer.trainable = True

# Re-compiling the model with a smaller learning rate
resnet_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-5),  # Using a smaller learning rate for fine-tuning
                     loss='categorical_crossentropy',
                     metrics=['accuracy'])

# Re-training the model with increased epochs and the learning rate scheduler
history = resnet_model.fit(
    train_generator,
    validation_data=valid_generator,
    steps_per_epoch=100,
    epochs=50,  # Increasing the number of epochs
    batch_size=64,
    validation_steps=50,
    callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3, verbose=1), lr_schedule]
)
```

```
Epoch 1/50
100/100 ━━━━━━━━━━━━━━━━━━━━ 441s 4s/step - accuracy: 0.3174 - loss: 2.3359 - val_accuracy: 0.0266 - val_loss: 3.8891 - learning_rate: 1.0000e-05
Epoch 2/50
100/100 ━━━━━━━━━━━━━━━━━━━━ 310s 3s/step - accuracy: 0.7551 - loss: 0.8703 - val_accuracy: 0.6369 - val_loss: 4.8596 - learning_rate: 1.0000e-05
Epoch 3/50
100/100 ━━━━━━━━━━━━━━━━━━━━ 356s 4s/step - accuracy: 0.7917 - loss: 0.7083 - val_accuracy: 0.0500 - val_loss: 9.9254 - learning_rate: 1.0000e-05
Epoch 4/50
100/100 ━━━━━━━━━━━━━━━━━━━━ 397s 4s/step - accuracy: 0.8078 - loss: 0.6192 - val_accuracy: 0.0419 - val_loss: 9.5092 - learning_rate: 1.0000e-05
Epoch 4: early stopping
```

Figure 35: Fine-tuning ResNet

- Figures 36, 37, and 38 show that the training and validation accuracy and loss are plotted, predictions are made on test data, and 30 random images are displayed along with the predictions and the model is saved.

```
# Plotting the training & validation Accuracy and Loss
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('ResNet50 Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('ResNet50 Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

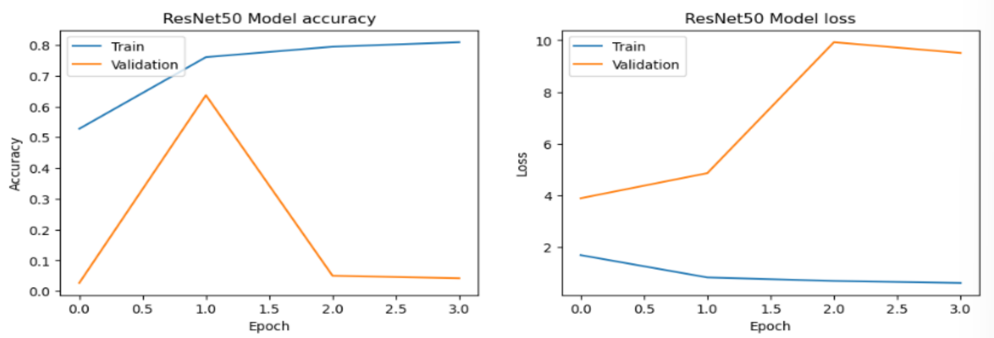


Figure 36: Plots for training and validation accuracy and loss

```
# Generating the predictions
test_generator = test_datagen.flow_from_dataframe(
    dataframe=test_df,
    directory=test_images_directory,
    x_col="file_name",
    y_col=None,  # No labels in the test set
    batch_size=64,
    seed=424,
    shuffle=False,
    class_mode=None,  # No labels
    target_size=(128, 128)
)
```

```
Found 153730 validated image filenames.
```

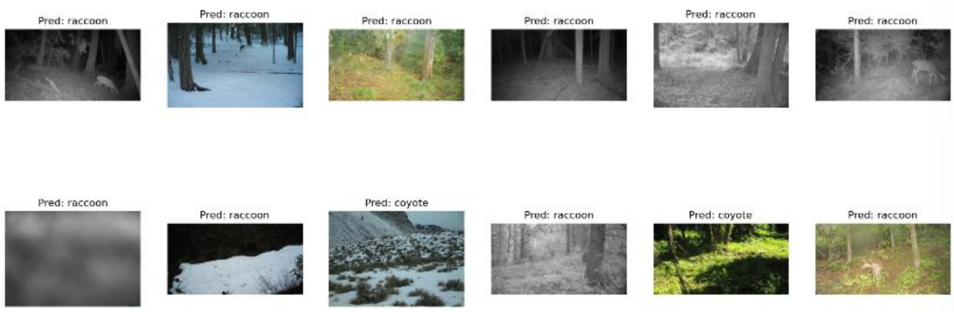
```
# Generating the predictions
predictions = resnet_model.predict(test_generator, steps=test_generator.n // test_generator.batch_size + 1)
predicted_classes = np.argmax(predictions, axis=1)
class_labels = list(train_generator.class_indices.keys())  # Use training classes for label names
```

```
C:\Users\Preena Rahul\anaconda3\Lib\site-packages\keras\src\trainers\data_adapters\py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should
call `super().__init__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass these argume
nts to `fit()`, as they will be ignored.
  self._warn_if_super_not_called()
2403/2403 ━━━━━━━━━━━━━━━━━━━━ 3667s 2s/step
```

Figure 37: Generating predictions

```
# Displaying predictions for random 30 images
random_indices = np.random.choice(len(predicted_classes), 30, replace=False)

# Display 30 random images with their predicted labels
plt.figure(figsize=(20, 20))
for i, idx in enumerate(random_indices):
    plt.subplot(5, 6, i + 1)
    img_path = test_generator.filepaths[idx]
    img = plt.imread(img_path)
    plt.imshow(img)
    plt.title(f"Pred: {class_labels[predicted_classes[idx]]}")
    plt.axis('off')
plt.show()
```




```
resnet_model.save('resnet_model.keras')
```

Figure 38: Displaying images with predictions and saving model

- The pre-trained models are loaded again and predictions are made on the validation set. The predictions are then combined using a simple weighted average ensemble. The accuracy of the ensemble model is calculated and displayed as shown in Figure 39.

```
efficientnet_model = load_model('efficientnet_model.keras')
inception_model = load_model('inception_model.keras')
resnet_model = load_model('resnet_model.keras')
```

```
print(os.path.exists('efficientnet_model.keras'))
print(os.path.exists('inception_model.keras'))
print(os.path.exists('resnet_model.keras'))
```

```
True
True
True
```

```
efficientnet_predictions = efficientnet_model.predict(valid_generator, steps=valid_generator.n // valid_generator.batch_size + 1)
inception_predictions = inception_model.predict(valid_generator, steps=valid_generator.n // valid_generator.batch_size + 1)
resnet_predictions = resnet_model.predict(valid_generator, steps=valid_generator.n // valid_generator.batch_size + 1)
```

```
614/614 ━━━━━━━━━━━━━━━━━━━━ 556s 898ms/step
614/614 ━━━━━━━━━━━━━━━━━━━━ 606s 985ms/step
614/614 ━━━━━━━━━━━━━━━━━━━━ 737s 1s/step
```

```
# Averaging the predictions
average_predictions = (efficientnet_predictions + inception_predictions + resnet_predictions) / 3

# Getting the final predicted classes
ensemble_predicted_classes = np.argmax(average_predictions, axis=1)

# True classes
true_classes = valid_generator.classes
class_labels = list(valid_generator.class_indices.keys())

# Calculating accuracy
ensemble_accuracy = np.sum(ensemble_predicted_classes == true_classes) / len(true_classes)
print(f'Ensemble model accuracy: {ensemble_accuracy}')
```

```
Ensemble model accuracy: 0.5510583560457475
```

Figure 39: Calculating Ensemble Accuracy

- The images are displayed randomly with the predictions as shown in Figure 40.

```
# Displaying random 30 predictions
random_indices = np.random.choice(len(ensemble_predicted_classes), 30, replace=False)

plt.figure(figsize=(20, 20))
for i, idx in enumerate(random_indices):
    plt.subplot(6, 5, i + 1)
    img_path = valid_generator.filepaths[idx]
    img = plt.imread(img_path)
    plt.imshow(img)
    plt.title(f"True: {class_labels[true_classes[idx]]}, Pred: {class_labels[ensemble_predicted_classes[idx]]}")
    plt.axis('off')
plt.show()
```

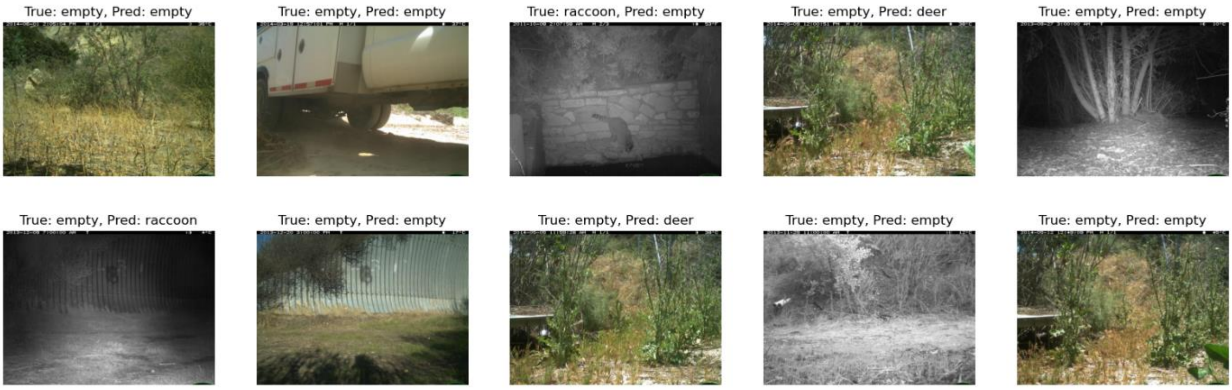


Figure 40: Displaying images with predictions

6 Section 6: Evaluation

- Figures 41, 42, 43, 44, and 45 show evaluation metrics for each model being displayed along with a code snippet. The confusion matrix, ROC curve, and classification report for each model are displayed.

```python
# Function to plot the confusion matrix
def plot_confusion_matrix(y_true, y_pred, class_labels, title):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(10, 8))
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title(f'{title} Confusion Matrix')
    plt.colorbar()
    tick_marks = np.arange(len(class_labels))
    plt.xticks(tick_marks, class_labels, rotation=45)
    plt.yticks(tick_marks, class_labels)

    fmt = 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
    plt.show()
```

```python
# Function to plot the ROC curve
def plot_roc_curve(y_true, y_score, n_classes, title):
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_true[:, i], y_score[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    plt.figure()
    for i in range(n_classes):
        plt.plot(fpr[i], tpr[i], label=f'Class {i} (area = {roc_auc[i]:.2f})')

    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.title(f'{title} ROC Curve')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend(loc='lower right')
    plt.show()


# Generating the evaluation metrics for each model and the ensemble
models = {
    'EfficientNet': (efficientnet_predictions, efficientnet_model),
    'InceptionV3': (inception_predictions, inception_model),
    'ResNet50': (resnet_predictions, resnet_model),
    'Ensemble': (average_predictions, None)
}
```

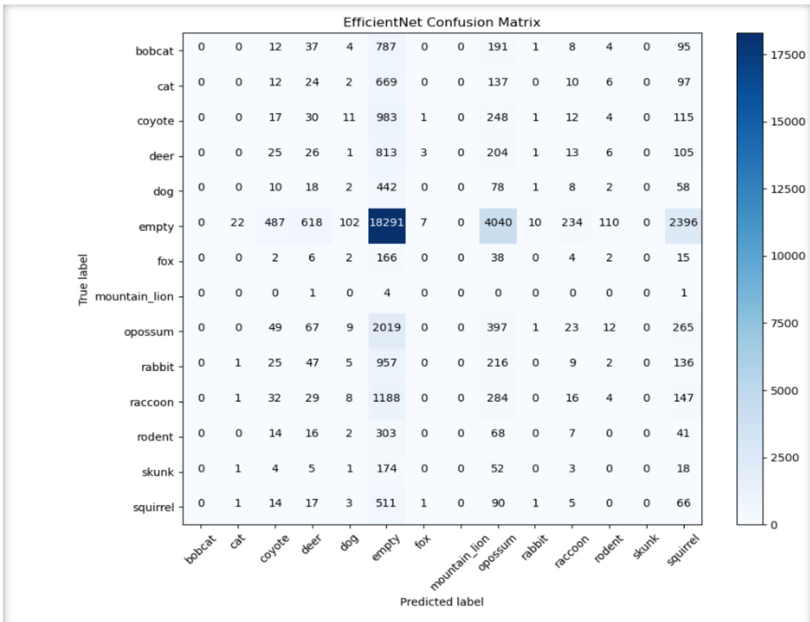
```python
for model_name, (predictions, model) in models.items():
    predicted_classes = np.argmax(predictions, axis=1)

    # Confusion Matrix
    plot_confusion_matrix(true_classes, predicted_classes, class_labels, model_name)

    # Classification Report
    print(f'{model_name} Classification Report')
    print(classification_report(true_classes, predicted_classes, target_names=class_labels))

    # ROC Curve
    y_true_binary = label_binarize(true_classes, classes=np.arange(len(class_labels)))
    plot_roc_curve(y_true_binary, predictions, len(class_labels), model_name)
```

Figure 41: Code snippet for plotting confusion matric, ROC Curve, and Classification report for all 4 models



EfficientNet Classification Report

	precision	recall	f1-score	support
bobcat	0.00	0.00	0.00	1139
cat	0.00	0.00	0.00	957
coyote	0.02	0.01	0.02	1422
deer	0.03	0.02	0.02	1197
dog	0.01	0.00	0.01	619
empty	0.67	0.70	0.68	26317
fox	0.00	0.00	0.00	235
mountain_lion	0.00	0.00	0.00	6
opossum	0.07	0.14	0.09	2842
rabbit	0.00	0.00	0.00	1398
raccoon	0.05	0.01	0.02	1709
rodent	0.00	0.00	0.00	451
skunk	0.00	0.00	0.00	258
squirrel	0.02	0.09	0.03	709
accuracy			0.48	39259
macro avg	0.06	0.07	0.06	39259
weighted avg	0.46	0.48	0.47	39259

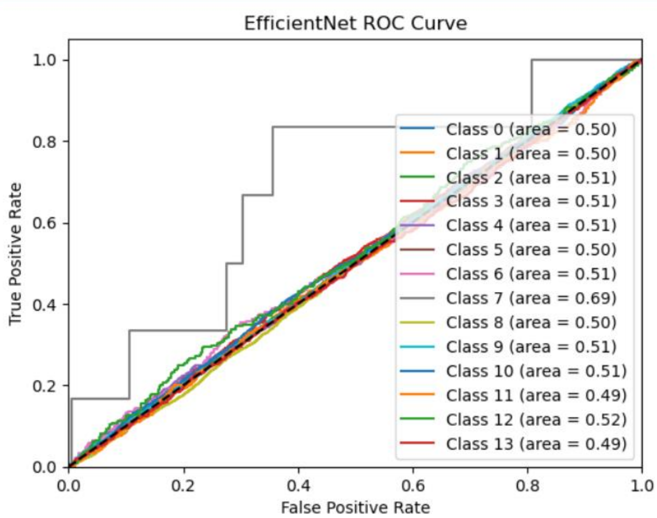
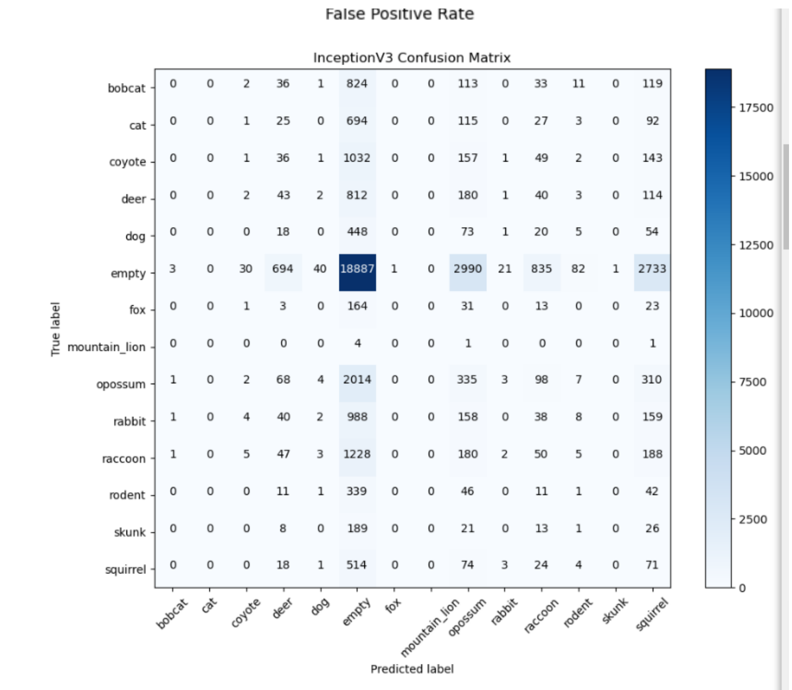


Figure 42: EfficientNet Evaluation Metrics



InceptionV3 Classification Report

	precision	recall	f1-score	support
bobcat	0.00	0.00	0.00	1139
cat	0.00	0.00	0.00	957
coyote	0.02	0.00	0.00	1422
deer	0.04	0.04	0.04	1197
dog	0.00	0.00	0.00	619
empty	0.67	0.72	0.69	26317
fox	0.00	0.00	0.00	235
mountain_lion	0.00	0.00	0.00	6
opossum	0.07	0.12	0.09	2842
rabbit	0.00	0.00	0.00	1398
raccoon	0.04	0.03	0.03	1709
rodent	0.01	0.00	0.00	451
skunk	0.00	0.00	0.00	258
squirrel	0.02	0.10	0.03	709
accuracy			0.49	39259
macro avg	0.06	0.07	0.06	39259
weighted avg	0.46	0.49	0.47	39259

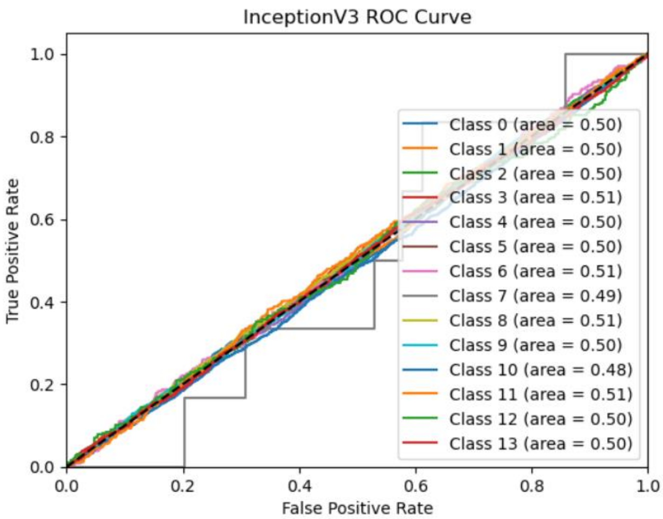
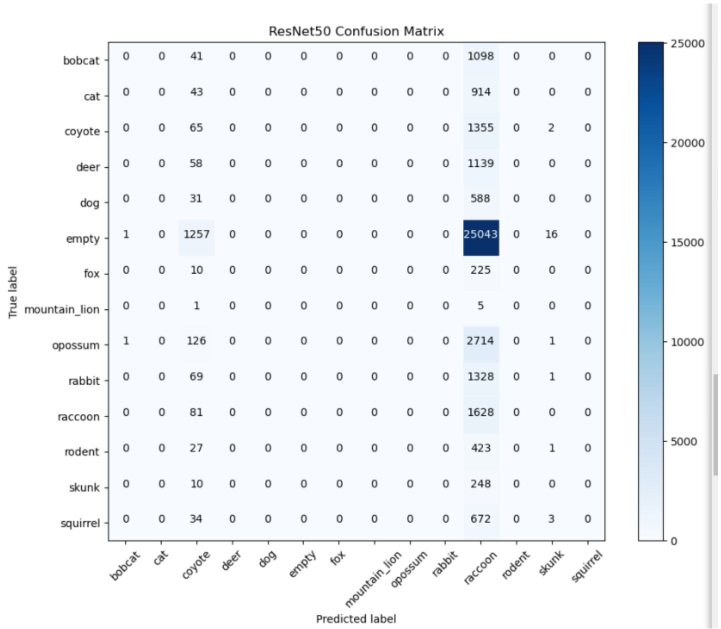


Figure 43: Inception Evaluation Metrics



Predicted label

ResNet50 Classification Report

	precision	recall	f1-score	support
bobcat	0.00	0.00	0.00	1139
cat	0.00	0.00	0.00	957
coyote	0.04	0.05	0.04	1422
deer	0.00	0.00	0.00	1197
dog	0.00	0.00	0.00	619
empty	0.00	0.00	0.00	26317
fox	0.00	0.00	0.00	235
mountain_lion	0.00	0.00	0.00	6
opossum	0.00	0.00	0.00	2842
rabbit	0.00	0.00	0.00	1398
raccoon	0.04	0.95	0.08	1709
rodent	0.00	0.00	0.00	451
skunk	0.00	0.00	0.00	258
squirrel	0.00	0.00	0.00	709
accuracy			0.04	39259
macro avg	0.01	0.07	0.01	39259
weighted avg	0.00	0.04	0.01	39259

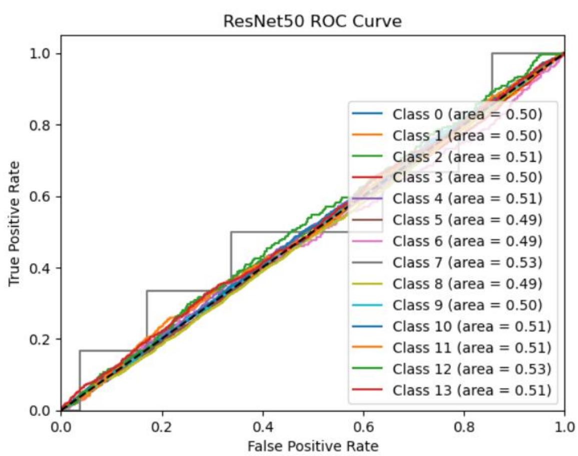
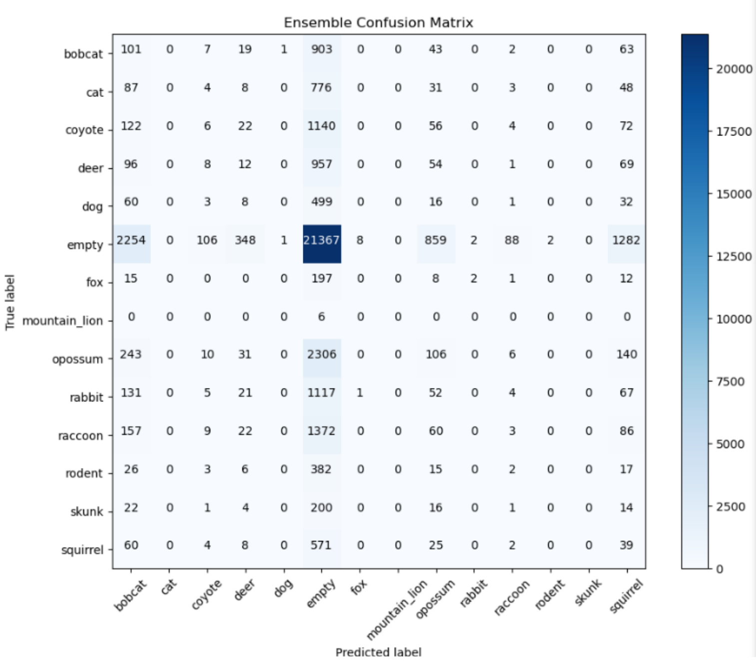


Figure 44: ResNet Evaluation Metrics



Ensemble Classification Report

	precision	recall	f1-score	support
bobcat	0.03	0.09	0.04	1139
cat	0.00	0.00	0.00	957
coyote	0.04	0.00	0.01	1422
deer	0.02	0.01	0.01	1197
dog	0.00	0.00	0.00	619
empty	0.67	0.81	0.74	26317
fox	0.00	0.00	0.00	235
mountain_lion	0.00	0.00	0.00	6
opossum	0.08	0.04	0.05	2842
rabbit	0.00	0.00	0.00	1398
raccoon	0.03	0.00	0.00	1709
rodent	0.00	0.00	0.00	451
skunk	0.00	0.00	0.00	258
squirrel	0.02	0.06	0.03	709
accuracy			0.55	39259
macro avg	0.06	0.07	0.06	39259
weighted avg	0.46	0.55	0.50	39259

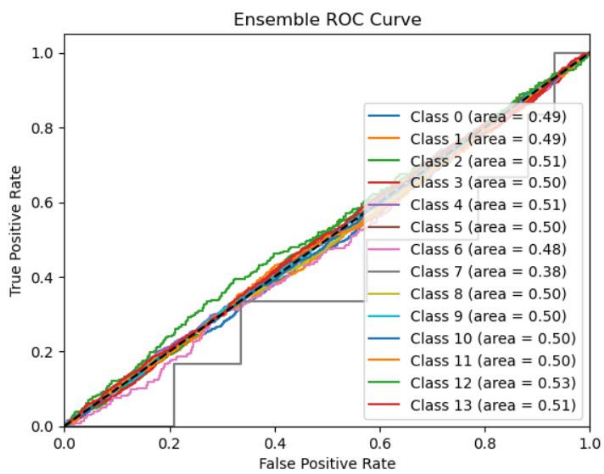


Figure 45: Ensemble model Evaluation metrics