UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Grado en Ingeniería del Software

# Animal identification using camera traps and detection models: Generation of a benchmark

# Identificación de animales mediante cámaras de fototrampeo y modelos de detección: Generación de un benchmark


Realizado por
Ziri Raha Ibnou-Cheikh

Tutorizado por
Cristóbal Barba González
Sandro Hurtado Requena

Departamento
Lenguajes y Ciencias de la Computación



MÁLAGA, junio de 2025

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

# Identificación de animales mediante cámaras de fototrampeo y modelos de detección: Generación de un benchmark

# Animal identification using camera traps and detection models: Generation of a benchmark


Realizado por
**Ziri Raha Ibnou-Cheikh**

Tutorizado por
**Cristóbal Barba González**
**Sandro Hurtado Requena**

Departamento
**Lenguajes y Ciencias de la Computación**

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2025



Fecha defensa: julio de 2025

# Abstract

Monitoring wildlife populations is essential for conservation and ecological research. Camera traps offer a non-invasive method to gather large volumes of image data, but manually reviewing this data is time-consuming and prone to error. This project proposes a complete pipeline for automating animal identification using object detection models. A large dataset comprising over 1.3 million images from the Sierra de las Nieves and Doñana natural parks was curated, cleaned, annotated, and split into training, validation, and testing sets. Several deep learning models—YOLOv8, Megadetector, and FasterRCNN—were trained and evaluated on this dataset. YOLOv8 achieved the best performance in terms of precision (93.4%) and bounding box accuracy (95.2% IoU), making it the most suitable for real-time deployment.

In addition to the dataset and model training, a web application was developed using Flask, PostgreSQL, Celery, and Redis. This platform allows users to run inference on new images, access the benchmark dataset, and request custom image sets. The entire system is containerized with Docker for scalability and ease of deployment. Designed with modularity and efficiency in mind, the solution serves both as a practical tool and a reproducible benchmark for future research in wildlife detection.

The full source code and dataset processing scripts are available on GitHub: https://github.com/ziriraha/snieves-donana-benchmark

**Keywords: Object detection, wildlife monitoring, camera traps.**

# Resumen

El monitoreo de las poblaciones de fauna silvestre es esencial para la conservación y la investigación ecológica. Las cámaras de fototrampeo ofrecen un método no invasivo para recopilar grandes volúmenes de datos de imágenes, pero la revisión manual de estos datos requiere mucho tiempo y es propensa a errores. Este proyecto propone un proceso completo para automatizar la identificación de animales mediante modelos de detección de objetos. Un gran conjunto de datos, compuesto por más de 1,3 millones de imágenes de los parques naturales de la Sierra de las Nieves y Doñana, se seleccionó, limpió, anotó y dividió en conjuntos de entrenamiento, validación y prueba. Varios modelos de aprendizaje profundo (YOLOv8, Megadetector y FasterRCNN) se entrenaron y evaluaron con este conjunto de datos. YOLOv8 obtuvo el mejor rendimiento en términos de precisión (93,4%) y precisión de detección (95,2% IoU), lo que lo convierte en el más adecuado para la implementación en tiempo real.

Además del entrenamiento del conjunto de datos y el modelo, se desarrolló una aplicación web utilizando Flask, PostgreSQL, Celery y Redis. Esta plataforma permite a los usuarios realizar detecciones sobre nuevas imágenes, acceder al conjunto de datos de referencia y solicitar conjuntos de imágenes personalizados. Todo el sistema está en contenedores con Docker para mayor escalabilidad y facilidad de implementación. Diseñada con modularidad y eficiencia, la solución sirve como herramienta práctica y como referencia reproducible para futuras investigaciones en detección de fauna silvestre.

El código fuente completo y los scripts de procesamiento del conjunto de datos están disponibles en GitHub:
https://github.com/ziriraha/snieves-donana-benchmark

**Palabras Clave: Detección de objetos, monitoreo de fauna silvestre, cámaras de fototrampeo.**

# Contents

# 1 Introduction

## 1.1 Motivation

Monitoring animal populations is essential for biodiversity conservation and ecological research. Camera traps are a non-intrusive way of collecting large volumes of data, but reviewing this data manually is time-consuming and error prone. Applying object detection models can automate this process and improve both speed and accuracy.

This project aims to generate a benchmark of models trained in the Sierra de las Nieves and Doñana dataset and provide researchers access to the model results and dataset. The goal is for researchers to train their own models and compare results in a standardized environment.

## 1.2 Objectives

The goals of this project are:

- Generate a clean and balanced dataset of wildlife images from camera traps.
- Train and compare several object detection models using this dataset.
- Develop a web application to run detections and generate custom datasets.
- Provide a benchmark in said web application to support future research in wildlife detection.

These objectives are approached with a focus on efficiency and scalability throughout the system.

## 1.3 Structure of the document

This document is divided into five chapters. The State of the Art explains the background and the evolution of object detection models and their datasets. Then in the methodology, the data preparation, training and application design are described. The development chapter explains the implementation details. And finally, the conclusion and future research lines are presented. In the appendix a manual of installation is provided.

## 1.4 Technologies used

The technologies used in this project are primarily Python[12] as it is the leading language used in the machine learning field. During the dataset processing, Pandas[14] was used for most of the tasks. The detection models trained are YOLOv8[6], Megadetector[1], and FasterRCNN[13].

Flask[4] was used for the application development as it is a lightweight and flexible Python web framework that allows quick and easy development of web apps. As the database system, PostgreSQL[11] was chosen for its robustness and scalability. A task system was developed using Celery[19] and Redis[18] for background task processing. Externally, a MinIO[9] server was used for storing and accessing image data. Finally, for containerization Docker[3] and Docker Compose[5] were chosen.

# 2

# State of the Art

The field of image classification and object detection has significantly evolved over the past decades. Early methods relied on handcrafted features and traditional statistical models, whereas modern approaches leverage deep neural networks, particularly Convolutional Neural Networks (CNNs).

## 2.1 Origins

In 1958, Frank Rosenblatt invented the Perceptron[16], one of the earliest artificial neural network models specialized for image classification. The Perceptron had an Input (pixels), Weights (adjustable parameters which determine the importance of each input), Activation Function (a function which determines the output), and an Output (a binary classification). The major problem with the Perceptron was that it lacked multiple layers and thus was restricted in its ability to learn complex patterns. This led to a decline in neural network research until the development of the Multi-Layer Perceptron (MLP) and the Backpropagation algorithm.

## 2.2 Multi-Layer Perceptron and Backpropagation

In 1986, a group of researchers—Rumelhart, Hinton, and Williams—introduced the Backpropagation[17] algorithm, which allowed efficient training of Multi-Layer Perceptrons (MLPs). MLPs overcame some of the limitations of the Perceptron by introducing one

or more hidden layers and non-linear activation functions, enabling them to learn more complex patterns.

## 2.3 Convolutional Neural Networks

In 1998 a major breakthrough happened, Yann LeCun introduced LeNet-5[8], the first ever CNN, used for digit recognition. Convolutional Neural Networks differentiate themselves from past approaches because of the following characteristics:

- Local Receptive Fields: Instead of connecting every input to every neuron, CNNs focus on small, localized regions of the input.
- Weight Sharing: The same filter (kernel) is applied across the entire image, reducing the number of parameters and helping generalization.
- Spatial Hierarchies: Stacked layers allow CNNs to learn low-level features in early layers (edges, textures) and high-level features in deeper layers (shapes, objects).
- Pooling Layers: These downsample feature maps to reduce dimensionality and computation, while retaining important spatial features.
- Translation Invariance: Because of how filters and pooling work, CNNs can recognize patterns even if they appear in different parts of the image.

These features made CNNs very powerful but they were limited by computational resources and lack of large-scale labeled data. In 2012 this changed with the introduction of AlexNet[7] by Krizhevsky, Sutskever, and Hinton. Which was trained on the ImageNet dataset[2]. AlexNet introduced key innovations like ReLU activation, dropout, and GPU training.

## 2.4 Datasets

The progress of deep learning in image classification and object detection has been strongly driven by the availability of large-scale annotated datasets. ImageNet, introduced in 2009, revolutionized the field by providing 14 millions of labeled images across 20,000+ categories, enabling the training of deep CNNs with high accuracy.

Beyond general-purpose datasets, domain-specific datasets have emerged to tackle challenges in particular environments. One notable example is the Snapshot Serengeti[20] dataset, a benchmark consisting of high-frequency annotated camera trap images captur-

ing 40 mammalian species in the African savanna. This dataset provides unique challenges such as varied lighting, occlusions, and species diversity in natural habitats, making it a valuable resource for wildlife monitoring and ecological studies.

Snapshot Serengeti enables researchers to develop and evaluate models for automatic animal detection and classification in real-world conditions. But the Snapshot Serengeti also highlighted some challenges such as class imbalance, small or partially visible animals, and variable lighting.

Recent research has proposed more targeted solutions for animal detection in the wild. A recent study[10] presents a custom deep learning pipeline aimed at improving detection performance in complex camera trap scenarios. This study focuses on the Sierra de las Nieves and Doñana datasets (the same dataset we are working on), which feature diverse species and Mediterranean ecosystems.

## 2.5 Conclusion

The progress in image classification and object detection owes much to the availability of large, annotated datasets. Public benchmarks like ImageNet and Snapshot Serengeti have played a critical role in driving innovation by providing shared resources for training, testing, and comparison. Making such datasets publicly accessible not only accelerates scientific research but also fosters collaboration, reproducibility, and the development of robust, specialized models.

# 3 Methodology

This project begins with collecting and preparing the large dataset of wildlife images, which involves cleaning, annotating, and formatting the data for training. From there, we train a range of object detection models, comparing their performance based on accuracy and speed. Finally, we integrate the best trained model into a web app that allows users to upload images and receive instant wildlife detections and request custom datasets.

## 3.1 Dataset

### 3.1.1 Introduction

We were given a very large dataset containing images from the Sierra de las Nieves and Doñana park, this dataset has 1,331,309 images. The species/classes that were of importance and their names are in Table 3.1.

After filtering the major dataset by keeping these classes we were left with 1,247,182 images to analyze (a reduction of 84,127 images).

### 3.1.2 Analysis

With such a large dataset it was important to analyze it. Some questions were arised: How many of these images are empty? Is there the same number of images per park? Is the amount of images per species balanced? This information is very important when training a detection model. Too little information about a species and the model will completely

| Label | Scientific name | Name |
|---|---|---|
| bos | *Bos primigenius* | aurochs / cattle |
| caae | *Capra aegagrus hircus* | domestic goat |
| caca | *Capreolus capreolus* | roe deer |
| can | *Canis lupus familiaris* | dog |
| capi | *Capra pyrenaica* | Iberian ibex |
| cer | *Cervus elaphus* | red deer |
| dam | *Dama dama* | fallow deer |
| emp | empty | empty |
| equ | *Equus ferus* | horse |
| fel | *Felis catus* | domestic cat |
| fsi | *Felis silvestris* | wildcat |
| gen | *Genetta genetta* | common genet |
| her | *Herpestes ichneumon* | Egyptian mongoose |
| lep | *Lepus granatensis* | Iberian hare |
| lut | *Lutra lutra* | otter |
| lyn | *Lynx pardinus* | Iberian lynx |
| mafo | *Martes foina* | beech marten |
| mel | *Meles meles* | badger |
| mus | *Mus or Apodemus* | mouse |
| ory | *Oryctolagus cuniculus* | rabbit |
| ovar | *Ovis orientalis aries* | sheep |
| ovor | *Ovis orientalis musimon* | mouflon |
| rara | *Rattus rattus* | black rat |
| sus | *Sus scrofa* | wild boar |
| vul | *Vulpes vulpes* | red fox |

**Table 3.1.** Species code, scientific name and common name of each class.

ignore it, too much information about one and the model will predict all images as that class (overfitting). To answer the first question: How many of these images are empty? The answer is more than half as seen in Figure 3.1. Most of these empty images we wont use in our training dataset.

The second question we had: Is there the same number of images per park? This information is important as differences in scenery can create biases by the detection model, this is also the reason why we use empty images, to prevent the background from having an effect on the prediction. In Figure 3.2 we can see that our dataset is somewhat balanced but Sierra de las Nieves has more images.

Our final question was: Is the amount of images per species balanced? As we can see in Figure 3.3, our dataset is extremely unbalanced, having hundreds of thousands of images in some species but only hundreds in others. This will be a challenge when training and designing our training dataset.

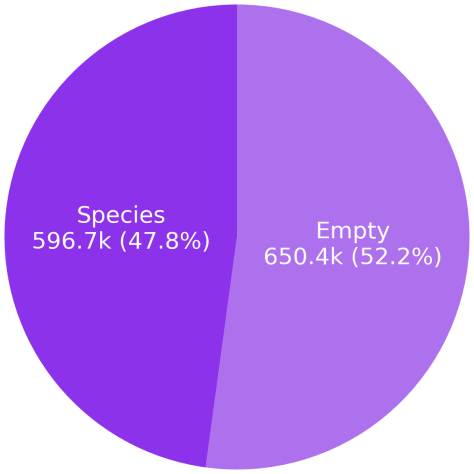



**Figure 3.1.** Comparison of Empty Images vs Species Images.

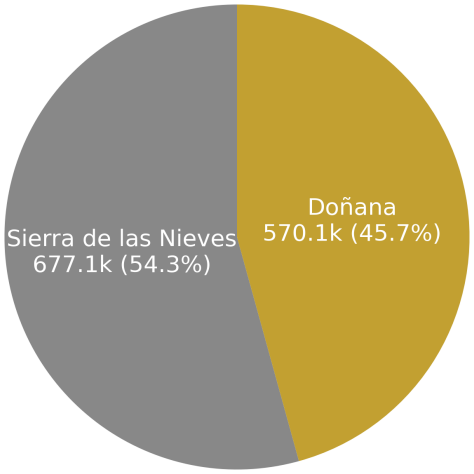



**Figure 3.2.** Comparison of amount of images by parks.

Knowing the balance of our dataset a new question arises: What's the species distribution in each of our parks? For Doñana (Figure 3.4) we can see that some species are missing from this park which is something we can expect. For Sierra de las Nieves (Figure 3.5) we can see that it has more variety in species than Doñana. As stated before, a balance between parks will help our detection model be more accurate and not have any biases related to the background of the image.

Based on the low number of images of certain classes, we can predict that these are going to be problematic and may yield worse performance.

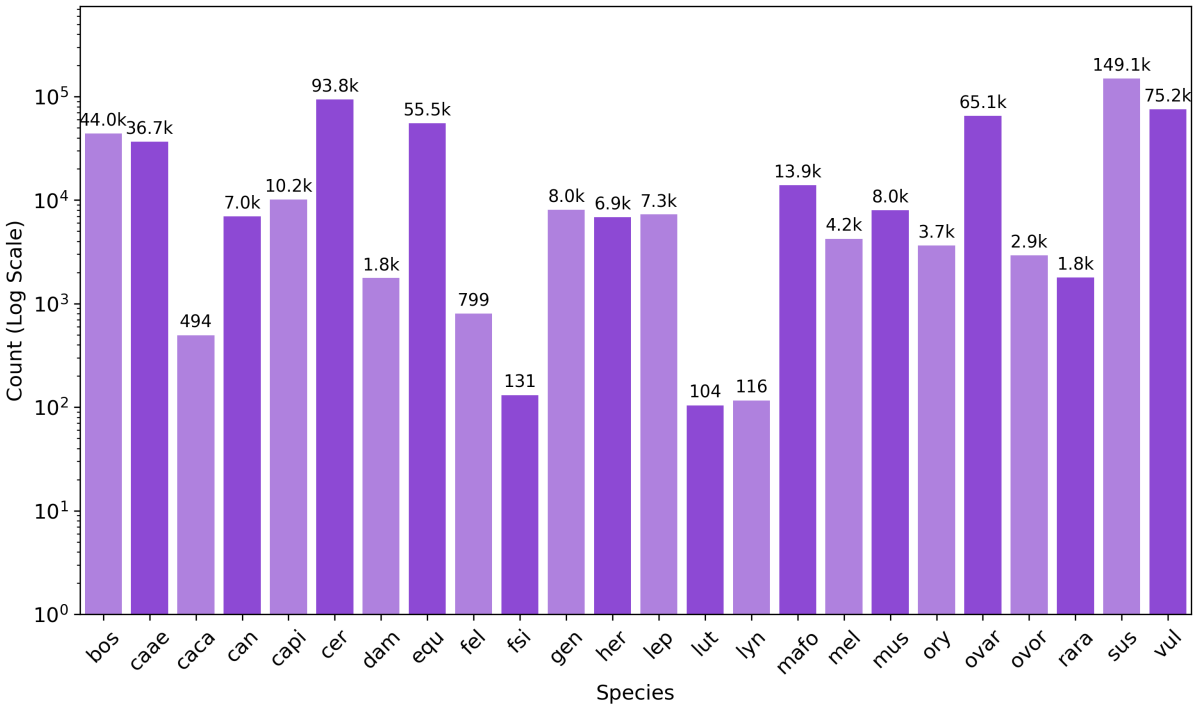



**Figure 3.3.** Distribution of number of images per species.

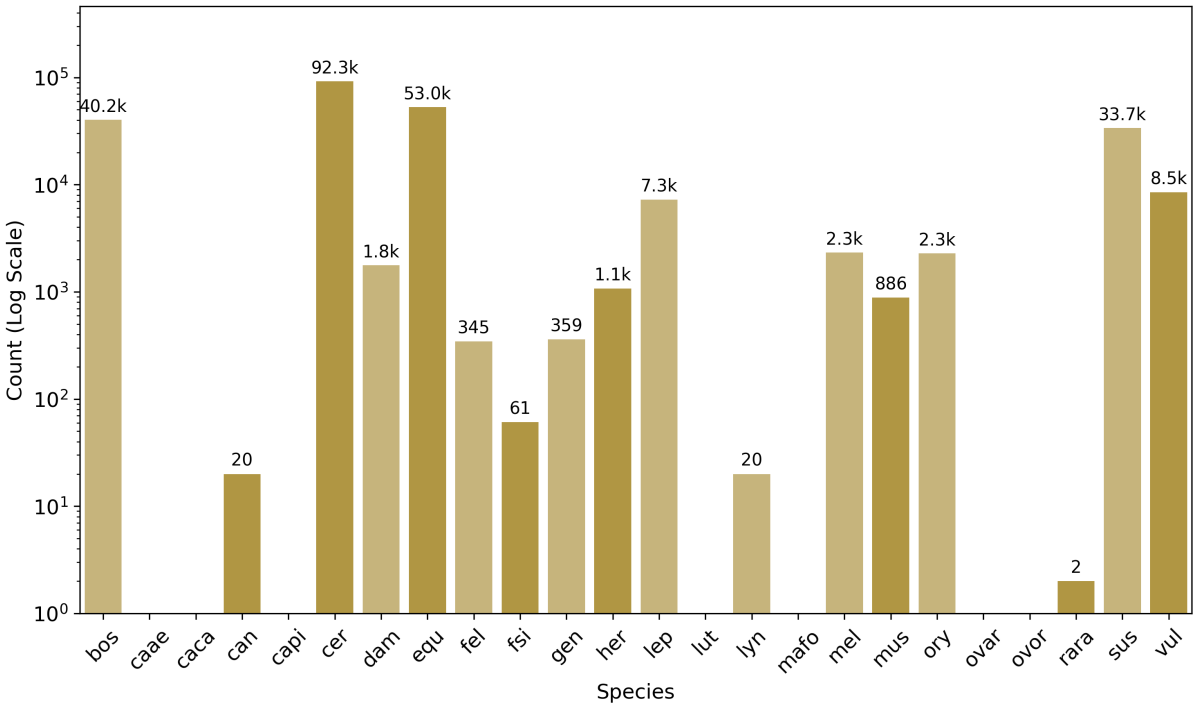



**Figure 3.4.** Distribution of number of images per species for the Doñana park.

### 3.1.3 Cleaning

We initally noticed that most of the pictures in the dataset were very similar. A lot of them were bursts, so most of the data was repeated. We decided to remove images taken

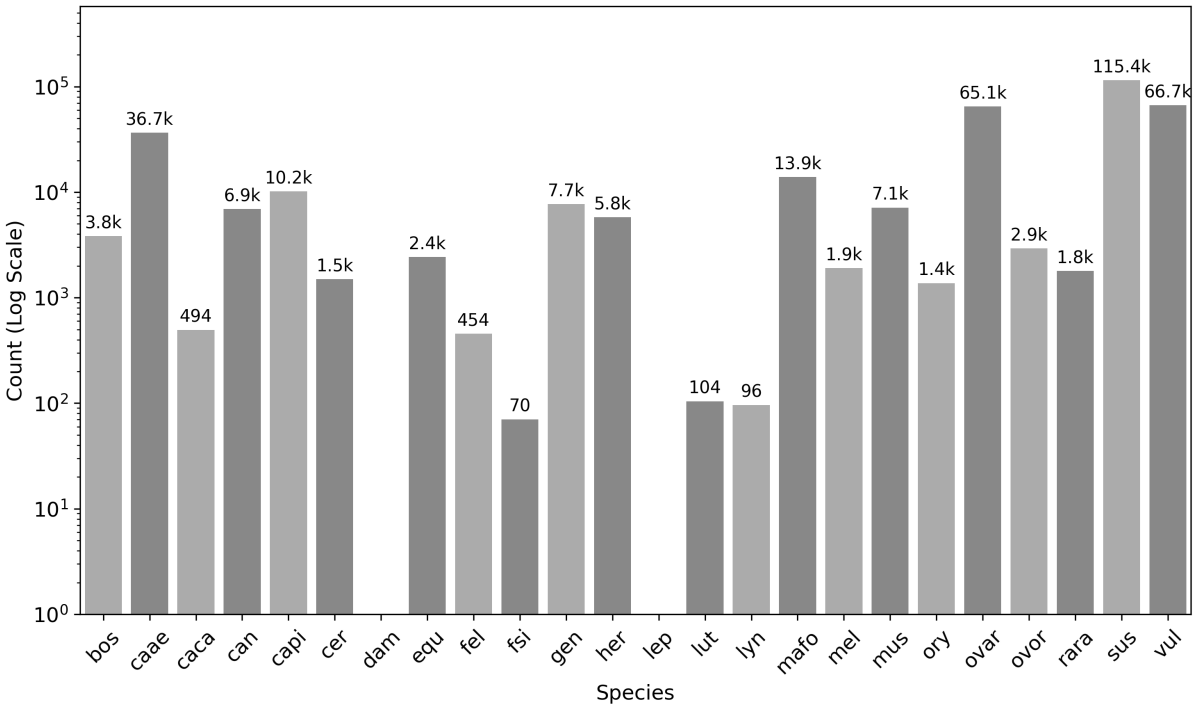



**Figure 3.5.** Distribution of number of images per species for the Sierra de las Nieves park.

less than a second apart, as seen in Figure 3.6 most images were discarded thanks to this rule.

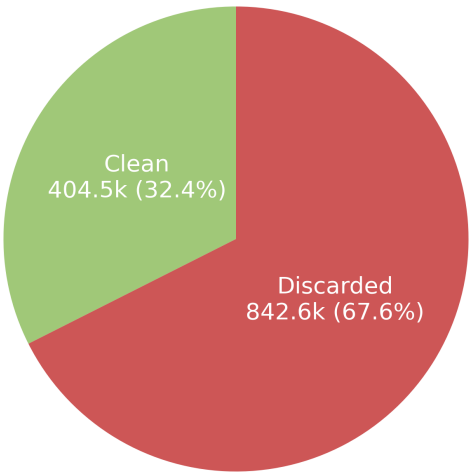



**Figure 3.6.** Comparison between clean images and discarded ones.

### 3.1.4 Split

A train, test and validation dataset split is needed for training. After cleaning, the images were split into two groups training-validation (80%) and testing (20%), the training-validation was further split into training (64%) and validation (16%). These groups were then balanced, as some parks had a lot more images than others, that could cause some overfitting.

As seen in Figure 3.7, the process of balancing gives us the following dataset split: Training 66% (75,835), Validation 13% (14,746) and Test 21% (24,697). 60.6% images were discarded in total from the dataset cleaning to the train-val-test split.

### 3.1.5 Annotations

For training our detecion model we need annotated data. More specifically we need the class or species of the animal in the image, and a bounding box of where in the image the animal is located. This annotation will be added in the form of a text file with the same name as the image containing the information. The class/species information must be presented in the form of a number, for that we have the following list as a guide, the index in which the species code is located in the list will be its number. Empty images are identified by not having an annotation associated to them.

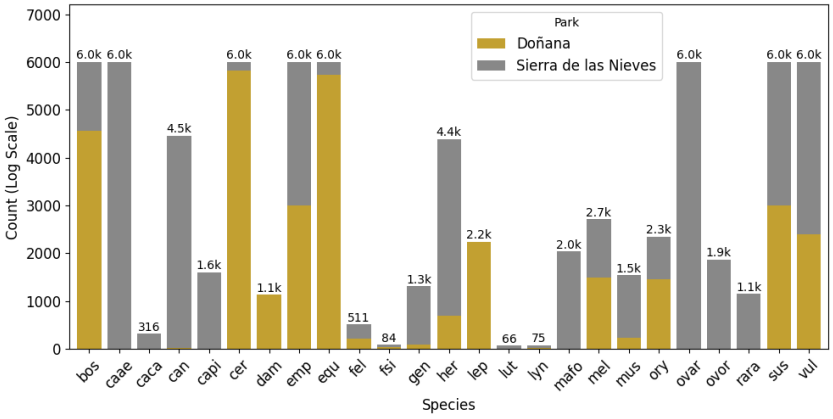
```
['bos', 'caae', 'caca', 'can', 'capi', 'cer', 'dam', 'equ', 'fel', 'fsi',
↪  'gen', 'her', 'lep', 'lut', 'lyn', 'mafo', 'mel', 'mus', 'ory', 'ovar',
↪  'ovor', 'rara', 'sus', 'vul']
```

#### Bounding Box

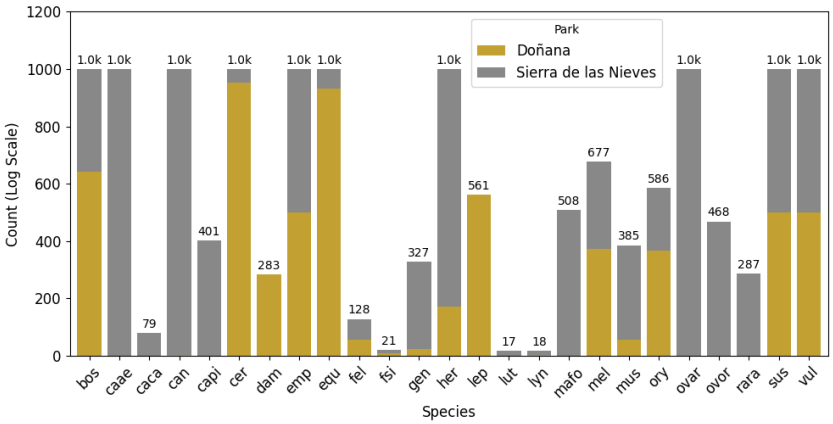
The provided images did not come annotated with bounding boxes. To address this issue, we will use the pretrained Megadetector model, chosen for its high precision, to generate bounding boxes for each image.

Instead of generating a bounding box for all the images as this process is very time consuming, only the images downloaded will generate their bounding boxes and save it. This way only the bounding boxes of the images we will use are generated.

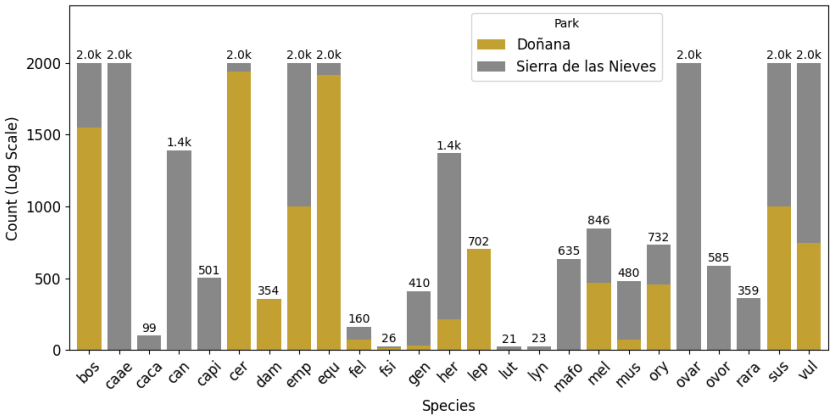
Megadetector generates the bounding box coordinates as the left-upper corner coordinates (x, y) and the height and width of the box, normalized to the image size. These




**(a)** Training Split




**(b)** Validation Split




**(c)** Testing Split

**Figure 3.7.** Species distribution by park for each dataset split.

coordinates were then converted to the YOLO format, which uses center coordinates along with the height and width of the box.

For example, for Figure 3.8 the following annotation would be generated:

```
18 0.66395 0.41476 0.5531 0.6925
```


**Figure 3.8.** An image of a sheep (Ovis orientalis aries).

### XML Annotation

One of the models we will later use for training is not compatible with the annotation format we use. We need to convert this text annotation into a special XML format, where the species/class is stated with its code and the bounding box is calculated differently, instead of being the center point, height and width of the image, it needs to be the left, right, top and bottom walls, this information is not normalized. The format of this XML annotation file for Figure 3.8 would be:

```
<annotation>
    <object>
        <name>ovar</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <occluded>0</occluded>
        <bndbox>
            <xmin>744</xmin>
            <xmax>1804</xmax>
            <ymin>74</ymin>
            <ymax>820</ymax>
        </bndbox>
    </object>
</annotation>
```

## 3.2 Model Training

### 3.2.1 Introduction

With the dataset obtained we are ready to train our model. In this case, we will train three models and use the best performing in our application. The goal for this model is to be fast and accurate, being able to run in very slow hardware like Raspberry Pis as the model ideally must be able to run in a camera trap too. We chose the following models for various reasons:

- YOLOv8 [6]: It is a state of the art image classification and detection model, its premise is that it runs the classification and detection at the same time, making it fast and accurate.
- Megadetector[1] [1]: It's an already trained model on an animal dataset, based on the YOLOv5 model. The theory here is that being a fine tuned model, its precision must be the greatest while performance will be similar to YOLOv8.
- FasterRCNN [15]: This model architecture was developed to put performance first. Also RCNN is known for great results.

### 3.2.2 Training

The three models will be trained with the same dataset and almost the same settings. The training will be done with 20 epochs, an image size of 640 pixels and a batch size of 64 images if each model allows it. The process for the three models is very similar, we start from a pretrained or base model and train it with our own data, this is called fine tuning.

### 3.2.3 Testing

The three models will be tested with the same dataset. To calculate the performance of a model we will use the following metrics:

- Precision.
- Recall.
- F1 Score.

[1]This model is the one used to generate the bounding boxes in the dataset.

- Average IoU (Intersection over the Union): The intersection over the union gives a metric in percentage of how accurate was the resulting bounding box.
- Accuracy.

The mAP can also give us a more detailed view on the bounding box score. We will also take a look at the Confusion Matrix of each model to analyze weaknesses and the F1-score across Species. The F1-score was chosen as the metric as it balances precision and recall and reflects overall performance better than accuracy in uneven datasets.

### 3.2.4 Results

As we can see in the Figure 3.9, the YOLO model outperformed both the Megadetector and FasterRCNN. We expected for the Megadetector model to take the lead in Precision as it is a fine tuned model and what is strange is that it performs worse in the detection even though the bounding box of the dataset was generated using it. FasterRCNN was the worst performing model and the slowest on training and testing, making it less than ideal for our use case.

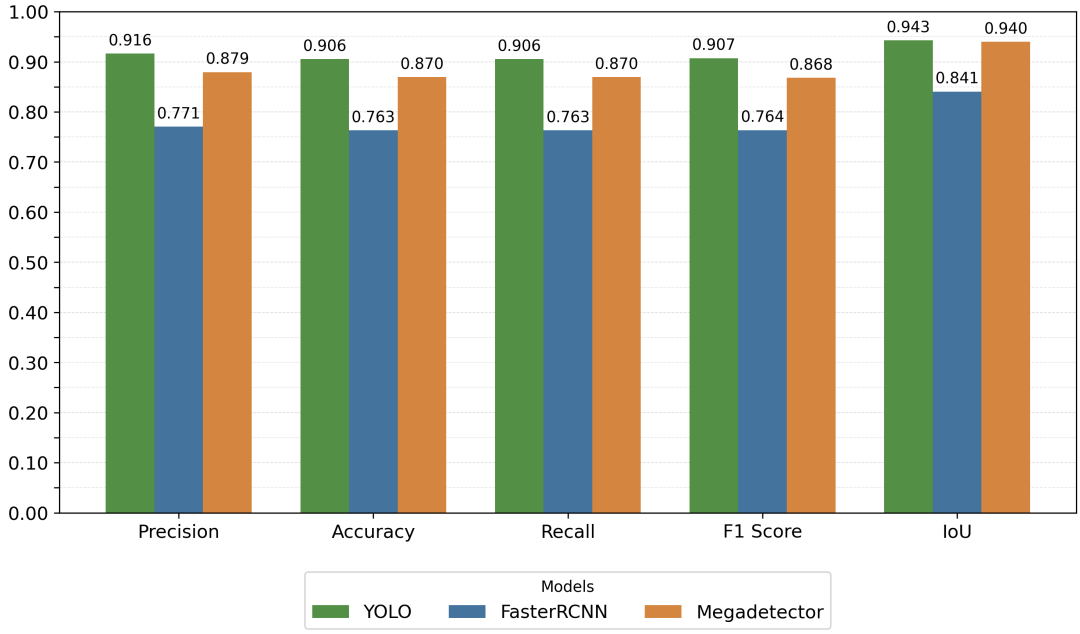



**Figure 3.9.** Metrics across models

Looking at the mAP across the models in Figure 3.10, we can see more clearly than in the Average IoU the performance of each model with the bounding box.

The winner in performance is YOLO and it may be due to being a very modern model (newer than Megadetector). In a previous test with a much smaller dataset the

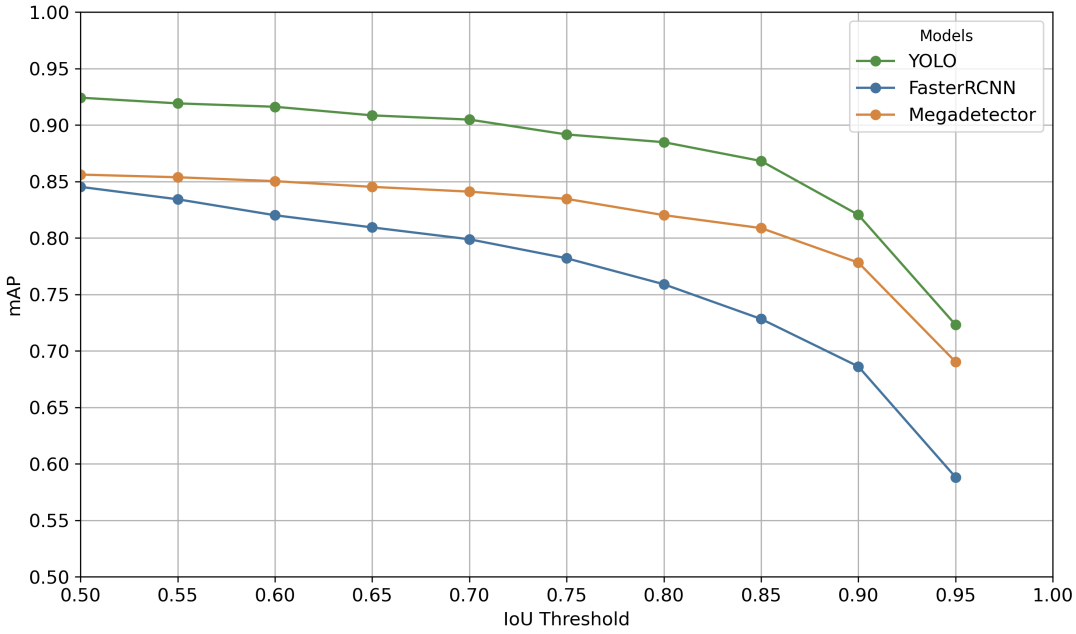



**Figure 3.10.** mAP across models

Megadetector and FasterRCNN models performed even worse with a precision of 60%. In theory, a larger dataset and/or more epochs could give us a better performance.

## 3.3 Web Application

### 3.3.1 Introduction

The web application to develop will be used to host the benchmark results and an API that enables researchers to apply detection to an image, or to download images from the dataset (generating custom datasets). The latter is very time consuming and resource heavy, as bounding boxes must be calculated for each image. Images are retrieved from a MinIO bucket and compiled into a ZIP file.

### 3.3.2 Technologies

The application will be developed in Flask. Flask is a lightweight Python web framework for building web apps and APIs. It's minimal, flexible, and lets you add only the components you need.

First we would need a database, images can be filtered for building a custom dataset and a database, more specifically, a relational database, allows us to store Image, Species

and Park information and filter/query them. A database will also allow us to save bounding box information to prevent having to calculate it more than once. We will use PostgreSQL for the database as it is a powerful relational database system known for reliability and extensibility.

The next thing we will need is a task queue, Flask cannot handle being blocked several minutes while it's performing a researcher's request. Celery is a distributed task queue for handling asynchronous background jobs in Python applications, it uses Redis as a message broker. We will use Celery for running tasks like downloading images or calculating bounding boxes, all in the background.

Finally, we will be using Docker to containerize the application, allowing us to package the Flask app, database, Celery workers, and Redis broker into isolated containers that communicate between each other.

### 3.3.3 System Architecture

Using Docker we will build the system shown in Figure 3.11. The Flask application will have an API module and a Views module. The API module will contain the logic for the API and the Views module will handle the HTML files to show to the users.

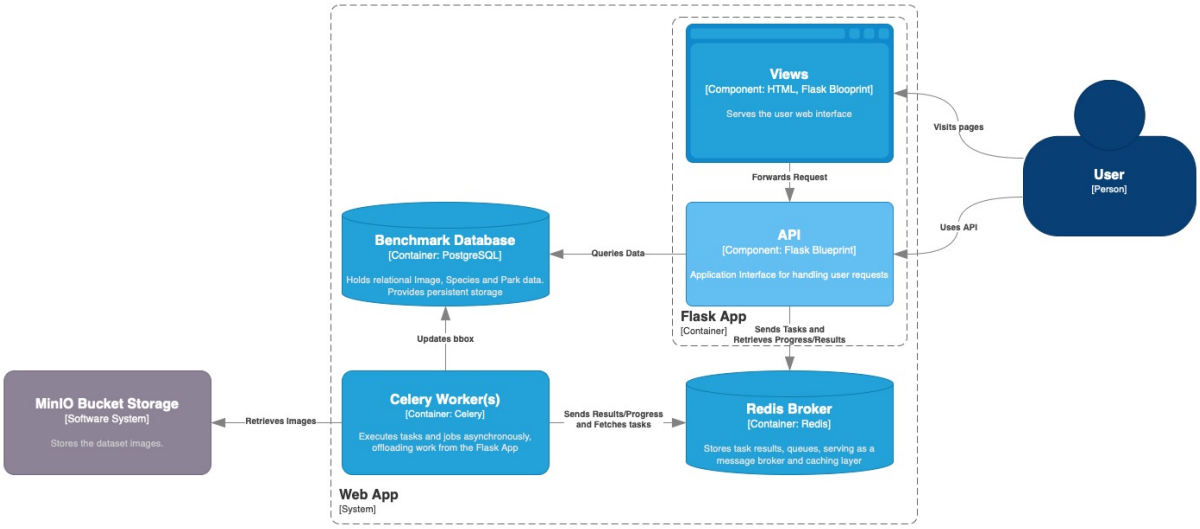



**Figure 3.11.** Container diagram of the web application.

When a request is received, the images to send to the user are queried from the Postgres database by the API module and sent to the Redis Broker for the Celery Worker(s) to process. Using Docker we can scale the Celery Workers as needed to boost performance. These Celery Workers will download the images from the MinIO Bucket storage (handled

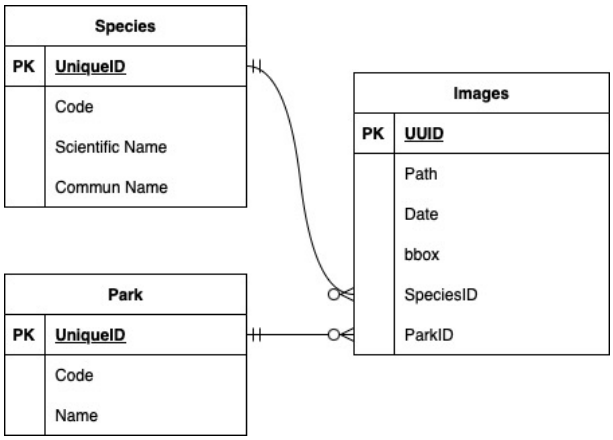



**Figure 3.12.** Database diagram for the web application's database.

externally) and compile a ZIP that will be saved in a shared Docker Volume for the API module to then send the user.

In the case of running detection on an image, the API module will send the task, which will be computed by a Celery Worker and the species information from the result will be queried from the database and returned to the user.

### 3.3.4 Database Design

We will use the Database to store Image, Species and Park information. For the image, we will need the following information:

- The MinIO path of the image.
- The bounding box of the image (if non empty).
- The Species/Class contained in the image.
- The park where this image was taken.

For the Species we need:

- Species code (eg. vul)
- The commun name (eg. Fox)
- The scientific name (eg. Vulpes vulpes)

For the Park we need:

- Park code (eg. donana)
- The name (eg. Doñana)

The codes are used for internal processing and as the names of the directories in the custom datasets, but for associating a Park or Species to an Image we will use an ID,

automatically generated. The name and scientific name will be used for showing the user. These relationships are illustrated in the database diagram (Figure 3.12).

# 4 Development

This chapter details the implementation of the dataset's utility scripts, the model training scripts and the development of the web application described in the Methodology. It covers the dataset cleaning and splitting, model training and detailed results, the structure of the web application, the development of both the API and user interface, the integration of background task processing, database interaction, and the containerization of all components using Docker.

## 4.1 Dataset

### 4.1.1 Introduction

As described in the Methodology, the dataset has gone through a cleaning to filter out repetitive images and was split to generate the training sets. We also cover how the images were annotated as no annotation was provided on the dataset.

### 4.1.2 Cleaning

For cleaning the dataset, first we needed to retrieve the date and time the images were taken in. A script was written to download the images without saving them and extract the date and time metadata:

```
def get_datetime_for_image(image_path):
    date = "nan"
    response = Downloader.get_image_from_minio(image_path)
```

```
    with Image.open(BytesIO(response.data)) as image:
        exifdata = image.getexif().get(306, None) or
        ↪  image.getexif().get(36867, None)

        decoded_exif = str(exifdata.decode('utf-8', errors='ignore') if
        ↪  isinstance(exifdata, bytes) else exifdata)
        decoded_exif = decoded_exif.replace('-', ':') if decoded_exif else
        ↪  None
        decoded_exif = decoded_exif.strip('\x00')

        date = datetime.strptime(decoded_exif, '%Y:%m:%d %H:%M:%S') if
        ↪  decoded_exif else None
        date = date.strftime('%Y-%m-%d %H:%M:%S') if date else "nan"
    response.close()
    response.release_conn()
    return date
```

To remove images taken less than a second apart, the `clean_bursts.py` script was written to go through this dataset with datetimes and discard the images that did not follow the rule. The classes with low image counts were excluded from this clearing (a low image class was determined to be a class with less than the median amount of images, this was done to prevent low image classes to lose all their data).

The script first does some formatting of the data and prepares it to be processed in parallel for better performance. Each group/class goes through the following function:

```
def process_group(group, time_interval=TIME_INTERVAL):
    dataframe = group.sort_values(by='date').reset_index(drop=True)

    prev_time = dataframe.loc[0, 'date']
    filtered = [dataframe.loc[0]]
    for _, row in dataframe.iterrows():
        time_diff = row['date'] - prev_time
        if time_diff.total_seconds() > time_interval:
            filtered.append(row)
            prev_time = row['date']
    return filtered
```

The function sorts the group's data by date and time and saves the previous date, if the current images has been taken more than 1 second after the previous images it passes the filter, if not it's discarded. The process continues until the end.

In Figure 4.1 we can visualize how many images were discarded per species, this cleaning helps balance the dataset by leveling the number of images we have per class.

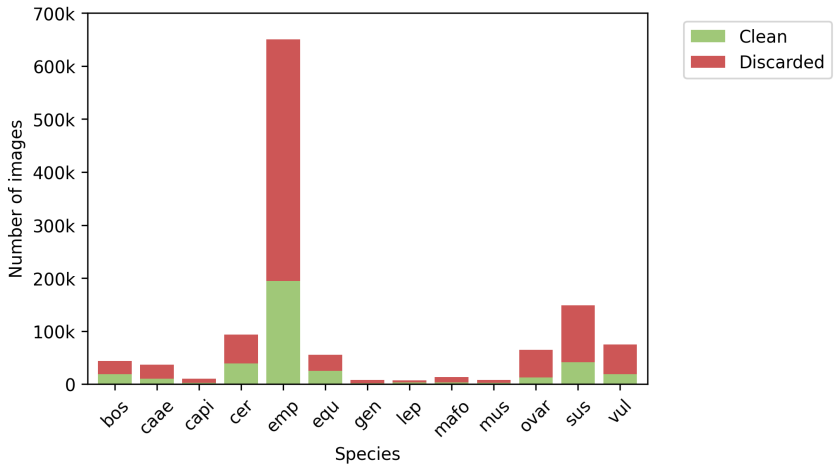



**Figure 4.1.** Clean and discarded images per species.

### 4.1.3 Splitting

During the splitting process, to prevent overfitting, the number of images per species was limited to the average number of images per park and species, rounded to the next thousand. A balanced distribution of images per species and park is ensured thanks to the `get_even_parks_per_species` function. For each species the following calculation is run:

```
def get_even_parks_for_species(images, top):
    total_images_in_parks = images.groupby('park').size().to_dict()
    num_parks = len(total_images_in_parks)
    if num_parks <= 1: return images.head(top)
    images_per_park = {park: min(top//num_parks, total_images_in_parks[park])
                       for park in total_images_in_parks}
    sorted_image_quantity = sorted(images_per_park.items(), key=lambda item:
    ↪  item[1])
    next_is_taking = top//num_parks
    for park, _ in sorted_image_quantity:
        images_per_park[park] = images[images['park'] ==
        ↪  park].head(next_is_taking)
        next_is_taking = top - images_per_park[park].shape[0]
    return pd.concat(images_per_park.values(), ignore_index=True)
```

The `top` variable will be the maximum amount of images per species we want, by default we use the rounded mean of all species, without counting empty images.

Then we calculate how many images per park there are, the idea is to balance the amount of images per species per park, by sorting the parks we know which one has the least amount of images and we take the maximum we can (up to `top` divided by the

number of parks which is two) and if there is less than `top/2` images, we compensate by including more images from the other park.

### 4.1.4 Annotation

As stated before, the dataset did not provide a bounding box, that needed to be calculated: The annotation generation was ran in a separate process to speed up the download of the images. The images were downloaded using separate threads, as it is an I/O-heavy operation, downloaded images then were added to a queue and were picked up by one of two workers that generate the annotation:

```
@staticmethod
def get_bbox(model, image):
    pred =
    ↪  model.generate_detections_one_image(image)['detections'][-1]['bbox']
    return f'{pred[0] + pred[2]/2} {pred[1] + pred[3]/2} {pred[2]} {pred[3]}'

def process_image(self, model, image_path, label_path, species):
    image = vis_utils.load_image(image_path)
    image.save(image_path, format='JPEG', quality=100)

    if self.detection and species != "emp":
        with open(label_path, 'w') as file:
            file.write(f'{self.CLASSES.index(species)} {self.get_bbox(model,
            ↪  image)}')
```

## 4.2 Model Training

### 4.2.1 Introduction

As explained in the Methodology Chapter, three models are going to be trained on the dataset: YOLO, Megadetector and FasterRCNN. Each model has a different way of performing an action, that is why each model will have its own procedure for training and testing.

### 4.2.2 Training

#### YOLO

The model used as a base was the pretrained YOLOv8s. The parameters used for training were the following: 20 epochs, batch of 16 and image size of 640. The training is run with

a python module:

```
model.train(data=DATASET_YAML, epochs=20, imgsz=640, batch=16, save_period=1)
```

#### Megadetector

The model used for training was the pretrained MD_5.0.0a. The parameters used for training are the following: 20 epochs, batch of 16, image size of 640 and Freeze of 20 layers. The command used for training was:

```
python yolov5/train.py --data DATASET_YAML --weights ./md_v5a.0.0.pt --epochs
↪  20 --batch-size 16 --imgsz 640 --freeze 20 --save-period 1
```

A freeze of 20 layers means that the 20 first layers of the model will not be changed. This was necessary as the model was too complex to train completely.

#### FasterRCNN

The pretrained "fasterrcnn_resnet50_fpn_v2"[13] model which is based on the Faster-RCNN model [15] will be used. The parameters used to train have changed due to hardware limitations: 20 epochs, batch of 4 and image size of 640. The command used for training was:

```
python fasterrcnn/train.py --data CONFI_DATASET_YAML --model
↪  fasterrcnn_resnet50_fpn_v2 --epochs 20 --batch 4
```

This model needed to use the XML Annotations mentioned in the dataset chapter. It was also the slowest model to train taking up to a week to finish the 20 epochs.

### 4.2.3 Testing

As the testing process is mostly the same across the models, a class was developed to handle the process and analyze the performance (it's located in the `utils.py` script):

To calculate the performance of a model we will only need three lists, one containing the true value of a classification attempt, the other the predicted value, and the third will be the intersection over the union (IoU) of both the real and predicted bounding boxes. The intersection over the union gives a metric in percentage of how accurate was the resulting bounding box.

In the class we have initialized the three lists and defined a function to run the testing process:

```
def run(self, get_pred):
    for img in os.listdir(self.images_folder):
        if not img.endswith('.jpg'): continue
        img_name = img.split('.')[0]
        rcls, rbbox = self.get_real(img_name)
        pcls, pbbox = get_pred(img_name)

        self.true.append(rcls)
        self.pred.append(pcls)
        # No IOU if empty image
        if rbbox and pbbox: self.iou.append(self.calculate_iou(pbbox, rbbox))
```

For each image in the testing data split, we will get the real value (true value), the predicted value and the IoU. If the image was classified as empty or is an empty image, we won't calculate the IoU as the output doesn't contain a bounding box.

The calculation of the real value is done via the `get_real` function:

```
def get_real(self, img_name):
    lbl_path = os.path.join(self.labels_folder, img_name + '.txt')
    if not os.path.exists(lbl_path): # Empty image
        real_cls = -1 # -1 will represent an empty image
        real_box = None
    else: # Non-empty
        with open(lbl_path, 'r') as f:
            clas, *bbox = f.readline().split()
            real_cls = int(clas)
            real_box = list(map(float, bbox))
    return real_cls, real_box
```

This function takes an image name and searches for it in the labels folder, if a label exists we read the information and output it, if a label does not exist. The image is an empty image and we return -1 (the numerical representation of an empty image).

The `get_pred` function is not defined in the class but inside each of the model's train-test scripts as each model has a different way of getting the results of the detection.

### YOLO

The YOLO testing is the easiest by far:

```
model = YOLO(SAVE_LOCATION)
def get_pred(img_name):
    pcls, pbbox = -1, None
    img_path = os.path.join(tester.images_folder, f'{img_name}.jpg')
    for result in model(img_path):
        if len(result.boxes) != 0:
            pcls = int(result.boxes.cls[0].item())
```

```
            pbbox = result.boxes.xywhn[0].tolist()
        break
    return pcls, pbbox
```

We initialize the model and for each image name, we find the image, run detection on it and retrieve the result with most confidence. If there is no box, it is classified as an empty image, if not we retrieve the values.

### Megadetector

For Megadetector, the process is different, as Megadetector does not provide a python module to run detection on custom models. We first need to run the detect command:

```
python3 detect.py --weights WEIGHTS_PATH --source IMAGES_PATH --max-det 1
↪  --device 0 --save-txt --nosave
```

The variables `$WEIGHTS_PATH` and `$IMAGES_PATH` are the paths of the model and of the test images respectively. We set the maximum number of detections to one and instruct the script to save the results in text files and to not save the image with the detection. This command saves the results of the detections in a `RUN_PATH` in text files named like the image. From there the process for checking the results is similar to the `get_real` function.

```
def get_pred(img_name):
    label = os.path.join(RUN_PATH, f'{img_name}.txt')
    if not os.path.exists(label): return -1, None

    with open(label, 'r') as f:
        clas, *bbox = f.readline().split()
        pcls = int(clas)
        pbbox = list(map(float, bbox))

    return pcls, pbbox
```

### FasterRCNN

FasterRCNN's process is similar to Megadetector in the sense of running detection (in this case called inference) on the images folder and retrieving the results from the run path. In this case, the results are saved in a csv file containing the image name and the detection.

```
python inference.py --input IMAGES_PATH --weights WEIGHTS_PATH --table --data
↪  CONFI_DATASET_YAML
```

The simple idea of checking the results of an image is to search the csv for that image and output the results. But that would take $O(n)$ every time we looked for a particular image name ($O(\log n)$ if we ordered the image names). In total, the running process of the tester would have a time complexity of $O(n^2)$. The best way to retrieve the results was to first parse the csv file and save its data into a dictionary which is a $O(n)$ operation. To later read the dictionary and search inside the dictionary which is a $O(1)$ operation. In total it would be a $O(n)$ process.

```
results = {}
with open(f"{RUN_PATH}/boxes.csv", "r") as file:
    for line in file.readlines()[1:]:
        img_name, pred_label, pred_xmin, pred_xmax, pred_ymin, pred_ymax,
        ↪  pred_width, pred_height, _ = line.split(',')
        with Image.open(os.path.join(IMAGES_PATH, f'{img_name}.jpg')) as img:
            size_x, size_y = img.size
        px = (float(pred_xmin) / size_x + float(pred_xmax) / size_x) / 2
        py = (float(pred_ymin) / size_y + float(pred_ymax) / size_y) / 2
        pw = float(pred_width) / size_x
        ph = float(pred_height) / size_y
        results[img_name] = (CLASSES.index(pred_label)-1, [px, py, pw, ph])
```

The format of the bounding box in the FasterRCNN model is different, so here we perform a conversion.

The `get_pred` function is the simplest one yet:

```
def get_pred(img_name):
    pcls, pbbox = -1, None
    if img_name in results:
        pcls, pbbox = results[img_name]
    return pcls, pbbox
```

### 4.2.4 Results

All of the metrics were calculated using the three lists (true values, predicted values and IoU) from the testing process and the `scikit-learn` python module which provides functions for calculating the metrics except for the mAP which was done manually.

#### YOLOv8s

The YOLO model performed extremely well with a precision of 93.4% and a Average IoU of 95.2% as shown on Figure 4.2.

In the confusion matrix on Figure 4.3 the model confuses some species but overall it's not bad.

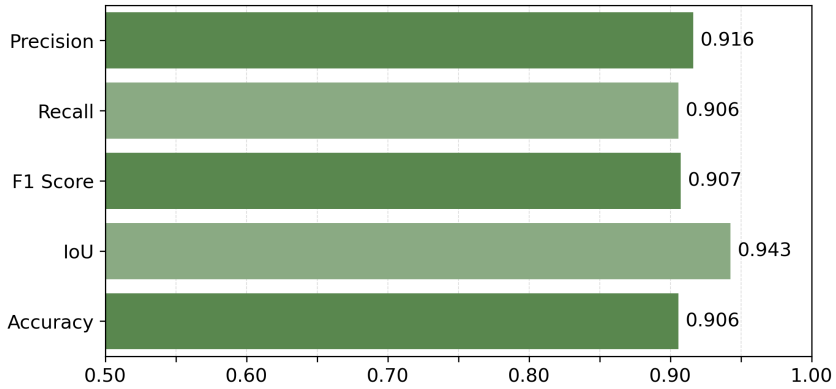



**Figure 4.2.** Metrics for YOLO

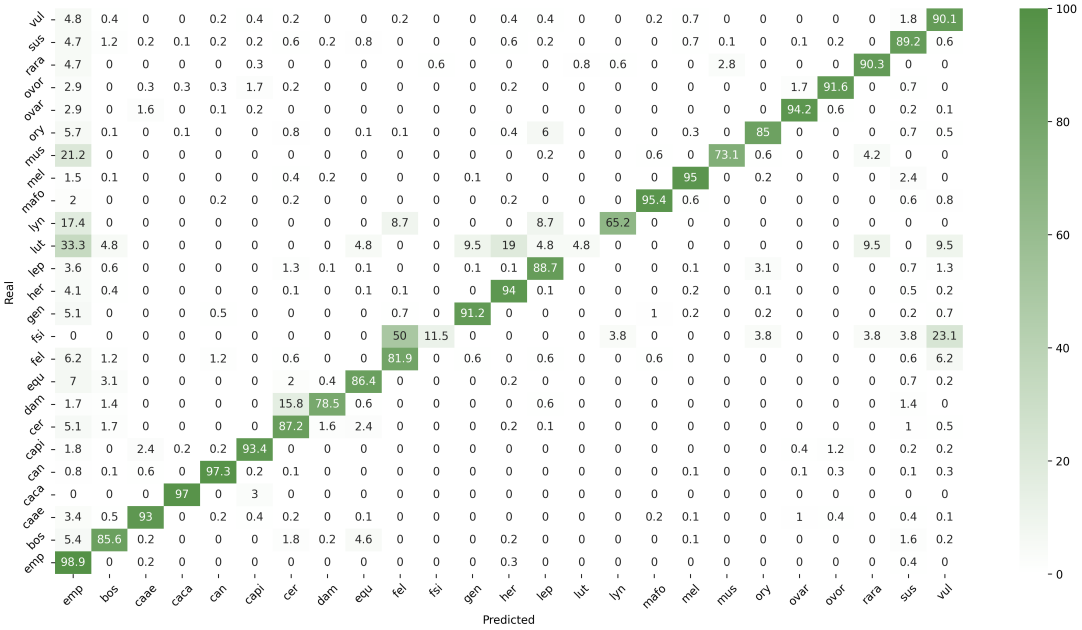



**Figure 4.3.** Confusion Matrix for YOLO

Looking at the F1 Score of each species on Figure 4.4, we can see a lower score in the wildcat (fsi) and otter (lut) which are some species where we have less support (images for training).

### Megadetector

The Megadetector model should have shown better results compared to YOLO on the detection as it was the model used to compute the bounding box and the layers responsible for it were frozen. Megadetector's performance was less than ideal, with a precision of 90.7% and an Average IoU of 94.6% as shown on Figure 4.5.

The confusion matrix, Figure 4.6, shows this lack of precision as in some classes there

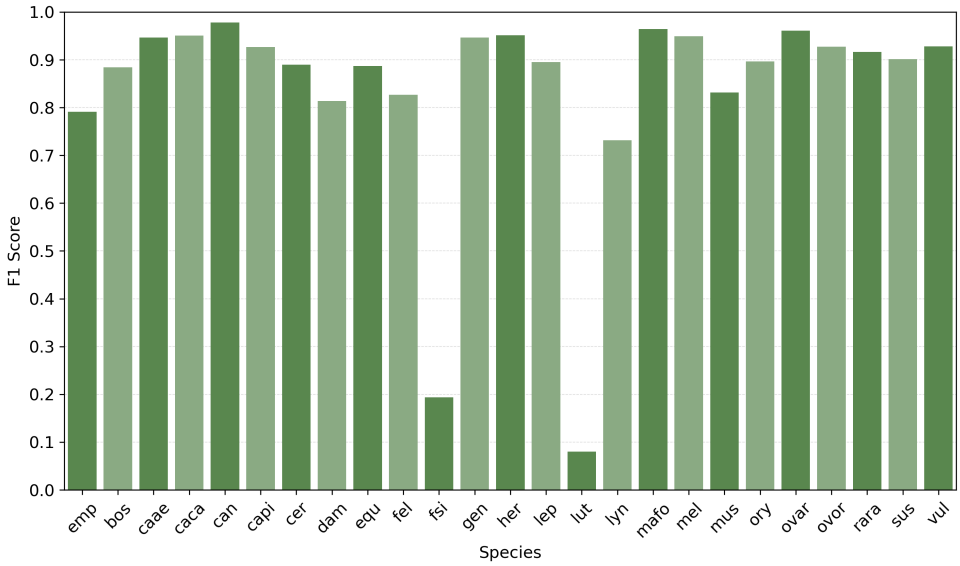



**Figure 4.4.** F1 Score across species for YOLO

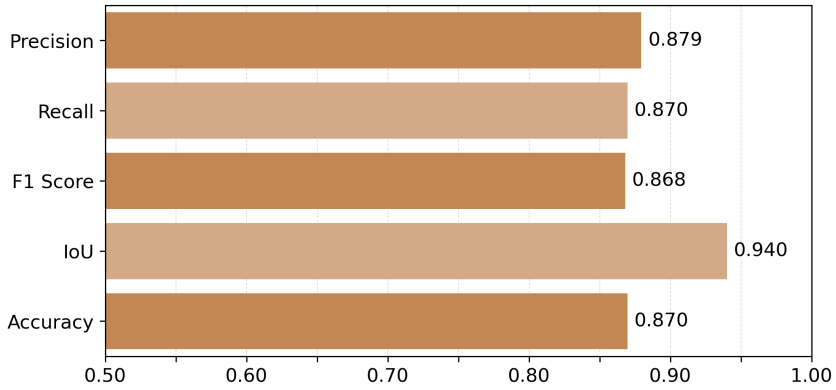



**Figure 4.5.** Metrics for Megadetector

are no correct predictions. We can also see some confusion between species, for example with the species of red deer (cer) and fallow deer (dam) which are very similar looking species.

With Megadetector we can clearly see the consequences on the lack of support in some classes with the Figure 4.7. The wildcat (fsi) and otter (lut) as they had a very small support were completely ignored by the model.

### FasterRCNN

The FasterRCNN model with a backend of ResNet50 should have shown better results due to its more complex prediction calculation. It has shown by far the worst results with a precision of 77.8% and an Average IoU of 84.8%, Figure 4.8.

On the confusion matrix, Figure 4.9, we have a better result than Megadetector but

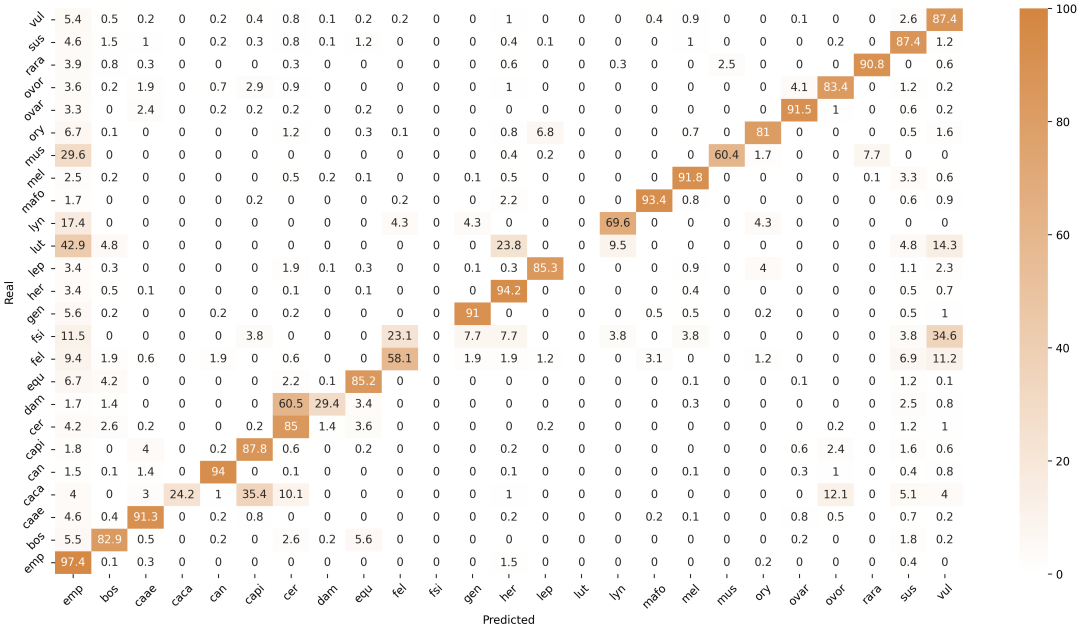



**Figure 4.6.** Confusion Matrix for Megadetector

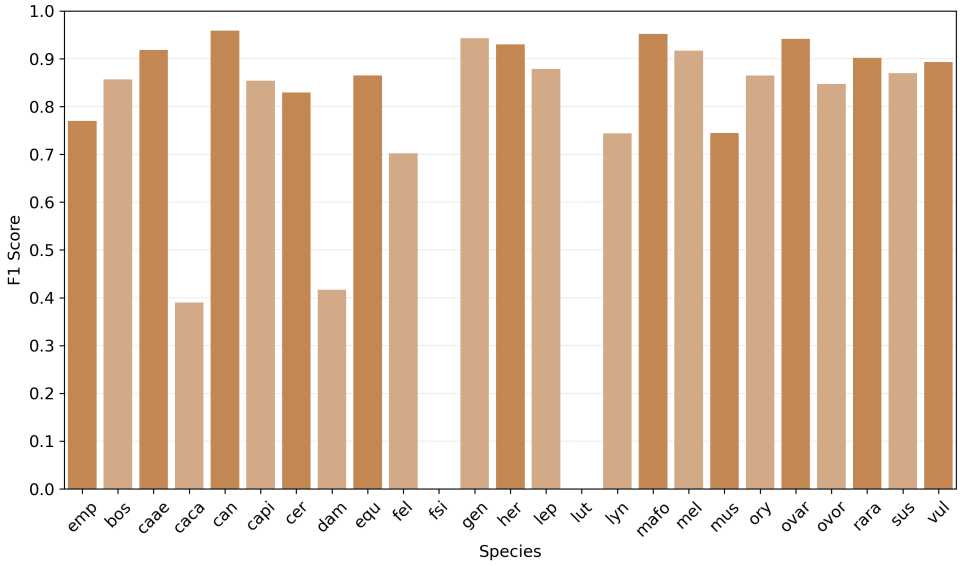



**Figure 4.7.** F1 Score across species for Megadetector

we can see that the errors are more spread out.

With the Figure 4.10 we can still see how much better FasterRCNN performed than Megadetector. Also, in the problematic species, the wildcat (fsi) and otter (lut), there is a better score than YOLO, but the fact that the rest of the species also have a poor score. This leaves YOLO being the best model.

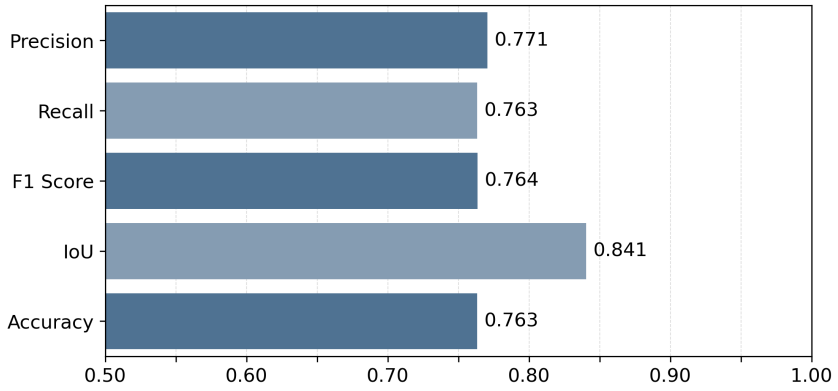



**Figure 4.8.** Metrics for FasterRCNN

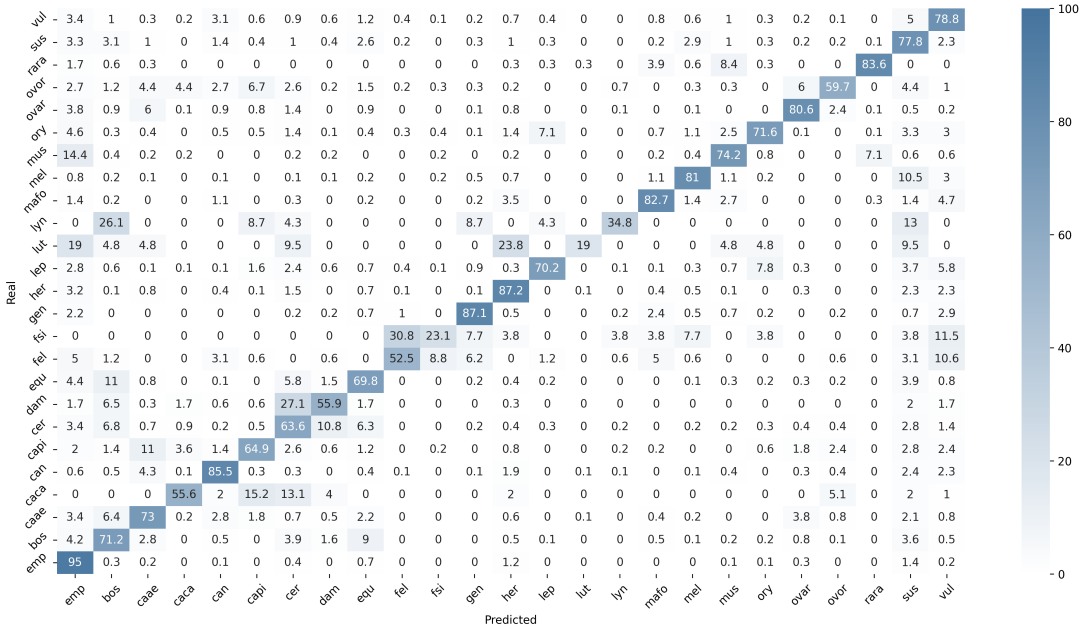



**Figure 4.9.** Confusion Matrix for FasterRCNN

## 4.3 Web Application

### 4.3.1 Introduction

In this section we cover the structure of the web application, the development of both the API and user interface, the integration of background task processing, database interaction, and the containerization of all components using Docker.

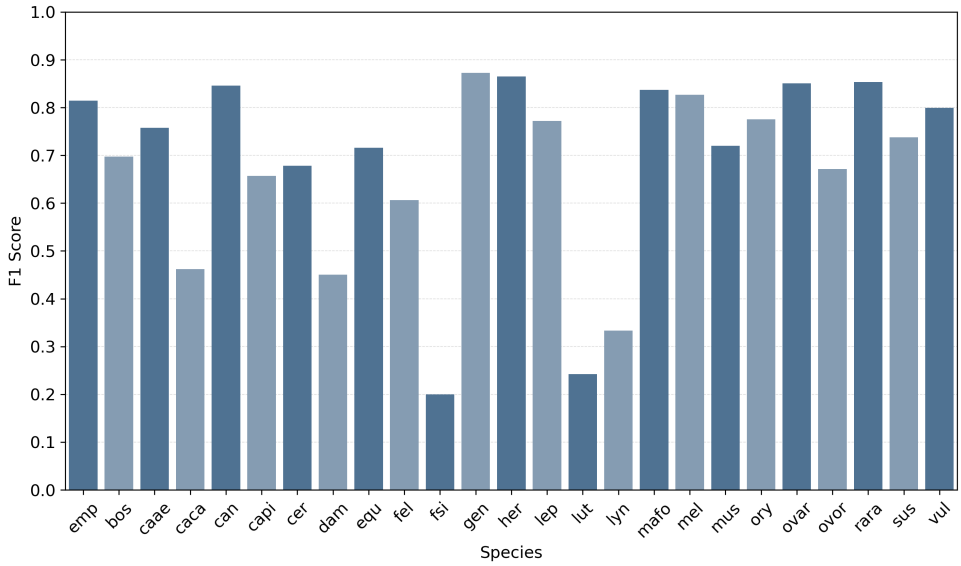



**Figure 4.10.** F1 Score across species for FasterRCNN

### 4.3.2 Project Structure

The Flask app was built using a factory pattern, which means the application instance is created inside a function (`create_app()` in `__init__.py`) instead of being declared globally. This approach makes the app more flexible, and modular.

The API logic is implemented in `api.py`, which defines the programmatic endpoints for interacting with the backend. The user-facing views, responsible for rendering HTML pages, are located in `views.py`. These files also define the HTTP routes and are imported in the app factory. Asynchronous background tasks, such as dataset requests and image detection, are handled in `tasks.py` using Celery.

Templates for rendering HTML are stored in the `templates/` directory. Configuration settings are centralized in `config.py` which imports some settings from an environment file (`.env`), and all third-party extensions (e.g. Celery, databases) are initialized in `extensions.py`. `constants.py` holds error message definitions and other constant values. And finally, `utils.py` file holds miscellaneous functions used by other components.

```
app/
   __init__.py
   api.py
   config.py
   constants.py
   extensions.py
   models.py
   tasks.py
   utils.py
```

```
views.py
static/
    images/
templates/
    api.html
    base.html
    benchmark.html
    dataset.html
    home.html
    inference.html
```

#### Database Integration

The database integration was done through SQLAlchemy which is a Python SQL toolkit and Object Relational Mapper (ORM). It allows us to define database tables as Python classes and write queries using Python syntax. The `models.py` file holds class declarations. Another useful feature of SQLAlchemy is the fact that it is able to create the tables in the database, simplifying the process of creating the database to only connecting it.

### 4.3.3 Frontend Views

Our application's views are set up using Jinja2 templates. We declared a `base.html` template from which the rest are built upon, this template links to the Bootstrap CDNs which are used for styling the pages and declares the navigation bar and the footer.

The pages we designed for the application are: a homepage (see Figure 4.11), a dataset page for making dataset requests and obtain information about it, a inference page for running inference on images, a benchmark page to view model's results and finally, an API page with documentation about the API endpoints.

The `dataset.html` file is the most complicated, it defines a HTML form which is used to execute a dataset request to the API endpoint, the result is processed to generate a progress bar (see Figure 4.12 and Figure 4.13).

### 4.3.4 API Implementation

In the `api.py` file is located the API logic. There are various endpoints available:

- `/api/datasets/<set_name>`: Download a specific dataset zip file by name (used to download the train, test or val datasets)
- `/api/species`: Get all species in the database.

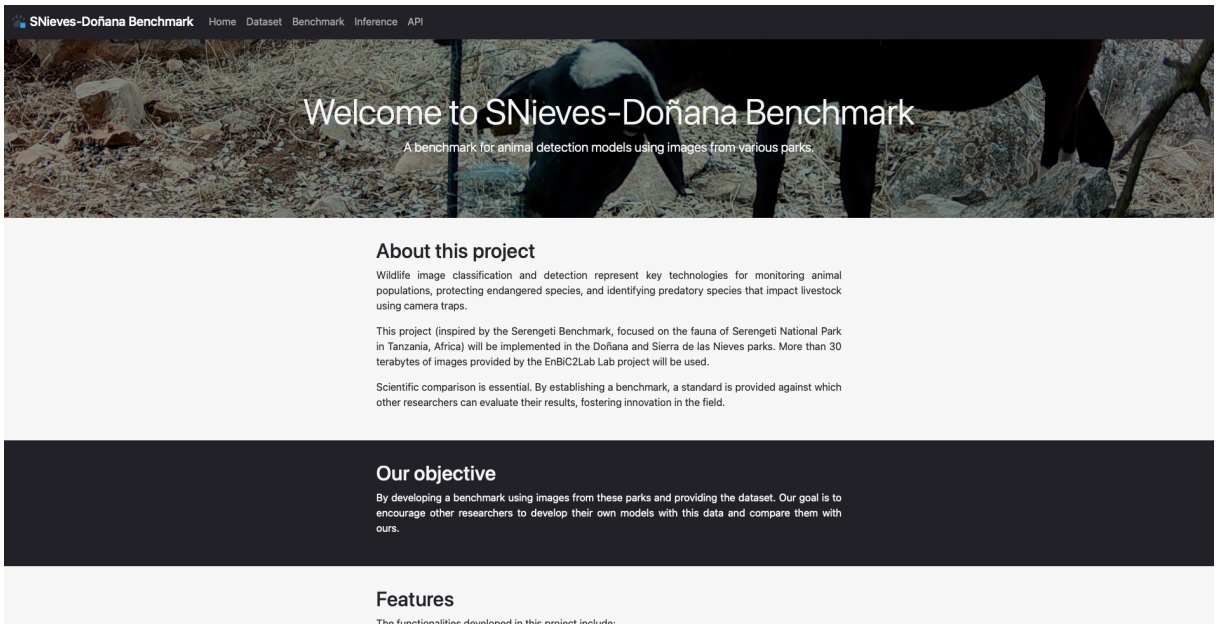



**Figure 4.11.** Screenshot of the web application's homepage.

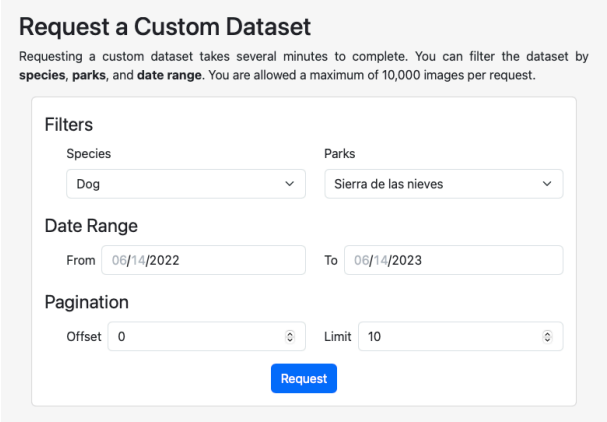



**Figure 4.12.** Form for making custom dataset requests.

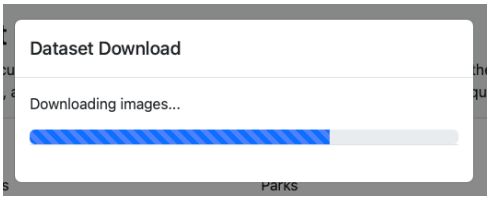



**Figure 4.13.** Progress bar for image download process.

- `/api/species/<scode>`: Get a specific species by its code (the idea is to be able to consult name or scientific name).
- `/api/parks`: Retrieve all parks in the database.
- `/api/parks/<pcode>`: Retrieve a specific park by its code (the idea is to be able to consult the name).
- `/api/images`: Query images with filters like park, species, and date (the idea is to

test the filters to use in the dataset request).

The dataset request logic was named "query". It was split into 3 different endpoints as when originally tested, a specific request could take up to several minutes to be completed:

- `/api/queries` (POST): Create a new ZIP generation job from filtered images (the same filters available in `/api/images` are available here).
- `/api/queries/<query_id>`: Check the status of a ZIP generation job.
- `/api/queries/<query_id>/download`: Download the result ZIP of a finished job.

When calling the first endpoint, the images are queried applying the filters and a list of images is passed to a task dispatched using Celery. A status message is also created preparing for the status cheking endpoint.

```
query = Image.query
if parks: query = query.filter(Image.park.has(Park.code.in_(parks)))
if species: query = query.filter(Image.species.has(Species.code.in_(species)))
if start_date: query = query.filter(Image.date >= start_date)
if end_date: query = query.filter(Image.date <= end_date)
images = query.order_by(Image.id).offset(offset).limit(limit).all()
if not images: return jsonify({'error': ERROR_NO_IMAGES_FOUND}), 404

task = generate_zip.delay([img.to_dict() for img in images])

redis_client.set(f'status:{task.id}', json.dumps({
    'status': 'Dispatched job... Waiting for it to start.',
    'progress': 0,
    'total': len(images)
}))
```

For status Redis was used as it works well with multiple services using the same data at once. When a user checks the status of a query, the information is fetched from Redis, the progress is calculated as it is slowly incremented by the celery task asynchronously. The task is also checked for failure or completion.

```
task = redis_client.get(f'status:{query_id}')
async_task = AsyncResult(query_id)
if not (task and async_task): return jsonify({'error': 'Query information not
↪  found.'}), 404

task_status = json.loads(task)
response = {
    "query_id": query_id,
    "status": task_status['status'],
    "progress": round(task_status['progress'] / task_status['total'] * 100)
}

if async_task.failed():
```

```
    redis_client.delete(f'status:{query_id}')
    response['failed'] = True
    response['error'] = async_task.result if async_task.result else 'An error
    ↪  occurred while processing the request.'
elif async_task.successful():
    redis_client.delete(f'status:{query_id}')
    response['completed'] = True
```

The dataset view uses this endpoint with an Interval to animate the progress bar. Once the request has been completed by Celery the user can use the `/api/queries/<query_id>/download` to download the resulting ZIP.

```
task = AsyncResult(query_id)
zip_file = task.get()
if not (zip_file and os.path.exists(zip_file)): return jsonify({'error':
↪  ERROR_GENERATE_ZIP}), 500

@after_this_request
def remove_zip(response):
    if os.path.exists(zip_file): os.remove(zip_file)
    redis_client.delete(f'status:{query_id}')
    return response

return send_file(zip_file, as_attachment=True,
↪  download_name=f'{query_id}.zip'), 200
```

After the download, the ZIP is deleted from the server to save storage.

Finally, the detection of animals in an image was named inference, inspired by the FasterRCNN model's naming:

- `/api/inference` (POST): Run species inference on an uploaded image.

Because the Redis Broker does not allow to pass files as arguments on tasks, the image sent in the body of the request is converted to base64 and then sent to a Celery task.

```
image = request.files.get('image')
if not image or not image.filename:
    return jsonify({'error': 'No image provided.'}), 400

bytes_image = BytesIO()
image.save(bytes_image)

base64_image = base64.b64encode(bytes_image.getvalue()).decode('utf-8')

task = inference_image.delay(base64_image)
try: result = task.get()
except Exception: return jsonify({'error': ERROR_INFERENCE}), 500

species = Species.query.filter_by(code=result['species_code']).first()
if not species: return jsonify({'error': ERROR_SPECIES_NOT_FOUND}), 404
```

```
return jsonify({'species': species.to_json(),
                'bbox': result['bbox'],
                'bbox_image': result['bbox_image']}), 200
```

The Celery task returns the species code, which is then used to search for the species information (name and scientific name) to show the user.

### 4.3.5 Task System

When a user requests a custom dataset or applies inference to an image, a Celery task is run asynchronously in the background, so the main app stays responsive and doesn't block while processing. The task system handles task queuing and more importantly, it can distribute tasks across workers to boost performance, making the app more scalable.

When a dataset request task is called, it calls the `generate_zip` function passing the images list. This function creates a temporary directory and uses the `download_images` function for downloading the images from MinIO and adding the annotations in that directory, once completed it generates a ZIP file which will be sent to the user.

```
@shared_task(bind=True)
def generate_zip(self, images):
    output_zip_file = None
    job_id = self.request.id
    redis_client.set(f'status:{job_id}', json.dumps({'status': 'Starting...',
    ↪  'progress': 0, 'total': len(images)}))
    try:
        with tempfile.TemporaryDirectory() as temp_dir:
            download_images(images, temp_dir, job_id)
            redis_client.set(f'status:{job_id}', json.dumps({'status':
            ↪  'Creating zip file...', 'progress': len(images), 'total':
            ↪  len(images)}))
            archive_file = os.path.join(app.config['API_DATA_DIRECTORY'],
            ↪  str(job_id))
            output_zip_file = shutil.make_archive(archive_file, 'zip',
            ↪  temp_dir)
    except Exception as e:
        logger.error(f"Error generating zip file: {e}")
        raise e
    redis_client.set(f'status:{job_id}', json.dumps({'status': 'Completed',
    ↪  'progress': len(images), 'total': len(images)}))
    return output_zip_file
```

The `download_images` function has been designed to download all images asynchronously in multiple threads improving performance. When a downloaded image has generated a bounding box, it is in this function where it saves it on the database.

```python
@shared_task
def download_images(images, destination, job_id=None):
    futures = []
    if job_id: redis_client.set(f'status:{job_id}', json.dumps({'status':
    ↪  'Downloading images...', 'progress': 0, 'total': len(images)}))

    with concurrent.futures.ThreadPoolExecutor(max_workers=
    ↪  app.config['MAX_CELERY_THREADS']) as executor:
        for image in images:
            save_path = os.path.join(destination, image['park']['code'],
            ↪  image['species']['code'])
            os.makedirs(save_path, exist_ok=True)

            filename = str(image['id'])
            image_path = os.path.join(save_path, f'{filename}.jpg')
            label_path = os.path.join(save_path, f'{filename}.txt')

            futures.append(executor.submit(download_image, image, image_path,
            ↪  label_path, job_id))

    for future in concurrent.futures.as_completed(futures):
        iid, bbox = future.result()
        if bbox:
            img = Image.query.get(iid)
            img.bbox = bbox
    db.session.commit()
```

The `download_image` function downloads the image from MinIO, checks if it requires a bbox, and if it's already generated, if not, it calls the `calculate_bbox` function to generate, it runs on a separate task being dispatched through the Redis broker. Redis does not allow objects to be sent, so the image is converted to base64.

```python
def download_image(image, image_path, label_path, job_id=None):
    new_bbox = None
    try:
        bytes_image = download_image_from_minio(image['path'], image_path)
        if image['species']['code'] != 'emp':
            if not image['bbox']:
                base64_image = base64.b64encode(bytes_image).decode('utf-8')
                new_bbox_task = calculate_bbox_wrapper.delay(base64_image)
                new_bbox = new_bbox_task.get(disable_sync_subtasks=False)
                image['bbox'] = new_bbox
            with open(label_path, 'w') as file:
                annotation_text = f"{image['species']['id']}
                ↪  {image['bbox']['x']} {image['bbox']['y']}
                ↪  {image['bbox']['width']} {image['bbox']['height']}"
                file.write(annotation_text)

    except Exception as e:
        logger.error(f"Error downloading image {image['id']}: {str(e)}")
```

```
        if os.path.exists(image_path): os.remove(image_path)
        if os.path.exists(label_path): os.remove(label_path)

    if job_id:
        progress = json.loads(redis_client.get(f'status:{job_id}'))
        progress['progress'] += 1
        redis_client.set(f'status:{job_id}', json.dumps(progress))
    return image['id'], new_bbox
```

This function was designed to run on the same worker as the `download_images` function because they share the local storage. To calculate the bounding box, we don't depend on the same file storage. So this function may run in different workers:

```
with vis_utils.load_image(BytesIO(bytes_image)) as image:
    detection = get_bbox_model().generate_detections_one_image(image)
    bbox = sorted(detection['detections'], key=lambda x: x['conf'],
    ↪  reverse=True)[0]['bbox']
    return {
        'x': bbox[0] + bbox[2]/2,
        'y': bbox[1] + bbox[3]/2,
        'width': bbox[2],
        'height': bbox[3]
    }
```

It uses Megadetector to generate the bounding box in the same process as the dataset.

When a user requests to running inference on an image, the detection model is retrieved and the detection is calculated, then the bounding box is drawn in the image and returned (see Figure 4.14 for an example). The result of the detection model is a number for the species, knowing the position of each class we search for that number in a list of classes.

```
def get_inference_calculation(bytes_image):
    model, pcls, pbbox, bbox_image = get_detection_model(), -1, None,
    ↪  BytesIO()
    with vis_utils.load_image(BytesIO(bytes_image)) as image:
        for result in model(image):
            if len(result.boxes) != 0:
                pcls = int(result.boxes.cls[0].item())
                pbbox = result.boxes.xywhn[0].tolist()
                dbbox = result.boxes.xyxy[0].tolist()
            break
        if pbbox:
            vis_utils.draw_bounding_box_on_image(image, dbbox[1], dbbox[0],
            ↪  dbbox[3], dbbox[2], use_normalized_coordinates=False)
            image.save(bbox_image, format='JPEG', quality='keep' if
            ↪  image.format == 'JPEG' else 95)
        else: bbox_image = None
    return INFERENCE_CLASSES[pcls] if pcls != -1 else 'emp', pbbox,
    ↪  bbox_image.getvalue()
```


**Figure 4.14.** Figure 3.8 with a bounding box.

When retrieving either model, in the `extensions.py` file we have defined the function to load the model if not loaded, or return it if already loaded:

```
bbox_model = None
def get_bbox_model():
    global bbox_model
    if bbox_model is None:
        bbox_model = run_detector.load_detector('MDV5A')
    return bbox_model
```

This prevents the models from being loaded in memory on workers that won't use them. Also allows the model to be garbage collected and free up the memory.

### 4.3.6 Containerization

Docker and Docker Compose were used for containerization. The web application relies on multiple components being started and docker compose simplifies the process of starting, managing, and linking these components (like the web server, Celery workers, Redis, and database) as isolated services in a single, unified environment.

There already exists container images for the Postgres database and the Redis broker, an image had to be created for the Flask application and Celery Workers:

```
FROM python:3.9.6

WORKDIR /app

COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt && \
    apt-get update && apt-get install -y libgl1 && \
    rm -rf /var/lib/apt/lists/*

COPY . .

# For megadetector
```

```
RUN git clone https://github.com/ultralytics/yolov5 && \
    pip install --no-cache-dir -r yolov5/requirements.txt
ENV PYTHONPATH="${PYTHONPATH}:/app:/app/yolov5"

EXPOSE 4000
CMD ["flask", "run", "--host=0.0.0.0", "--port=4000", "--debug"]
```

These components rely on Python 3.9.6 to run, this Dockerfile installs the requirements and also clones the ”yolov5” repository, needed for the Megadetector model. It also exposes the port 4000, where our application will be hosted.

In the docker-compose file, a shared volume was created for the Celery worker and Flask application, this volume is the one used for saving and collecting ZIP files.

```
services:
  app:
    container_name: app
    build: .
    command: ["hypercorn", "-c", "config.toml", "asgi:app"]
    env_file:
      - .env
    ports:
      - "4000:4000"
    volumes:
      - api_data:/app/data
    depends_on:
      - db
      - celery

  db:
    container_name: db
    image: postgres
    env_file:
      - .env
    volumes:
      - postgres_data:/var/lib/postgresql/data

  celery:
    build: .
    env_file:
      - .env
    command: celery -A make_celery worker --loglevel INFO
    volumes:
      - api_data:/app/data
    depends_on:
      - redis
      - db

  redis:
    container_name: redis
```

```
    image: redis

volumes:
  postgres_data:
    driver: local
  api_data:
    driver: local
```

The flask application command is modified in this file as for production environment hypercorn is used. Hypercorn allows the application to handle requests asynchronously. Also the Celery worker needs a special command to run. These two commands required special python files to run them.

By not exposing the database or redis ports, we make sure they are only reachable inside the docker compose environment making them more secure.

### Scalability

The web application is scalable, meaning more flask workers can be created using the `config.toml` file, these workers enable the application to handle multiple simultaneous requests. More Celery workers can also be summened using the docker compose scale feature and the command:

```
docker compose up --scale celery=3
```

This command creates three Celery workers for example.

# 5

# Conclusions and Futures Lines of Research

## 5.1 Conclusions

This project presents a complete pipeline for automated wildlife identification using camera trap images, combining dataset preparation, model evaluation, and web application development.

Starting with a raw dataset of over 1.3 million images from two natural parks, a rigorous filtering and cleaning process was applied. This included removing near-duplicate images, balancing the dataset across parks and species, and generating annotations using the Megadetector model. The final dataset was split into training, validation, and testing sets with attention to class balance and park diversity.

This dataset was used to train three object detection models: YOLOv8, Megadetector, and FasterRCNN. Due to low image quantity in the Wildcat and Otter classes, the three models showed a very bad performance trying to detect those species. Overall, YOLOv8 showed the best overall performance with a precision of 91.6%.

Beyond model training, a complete web platform was developed using Flask. The platform includes an API and user interface for running species detection using the YOLOv8 model on uploaded images, generating custom datasets, and downloading the dataset

splits used for training. It integrates Redis and Celery for asynchronous task management, PostgreSQL for structured data, and Docker for deployment. The design focuses on modularity and scalability, allowing it to support heavy workloads and multiple users.

In conclusion, this project not only delivered a usable animal detection model but also a practical tool for researchers. By enabling easy access to the data and allowing comparison of results, the platform fosters and encourages contributions to the field.

## 5.2 Future lines of Research

There are several directions in which this work can be expanded:

- Model Improvement: Explore newer architectures or train models for longer to improve detection performance, especially for underrepresented species.
- Multi-object Detection: Extend the system to handle images with multiple animals, enabling multi-label classification and bounding box generation for all present animals.
- Mobile and Edge Deployment: Develop a platform for small devices like the Raspberry Pi, enabling real-time, on-site inference in remote locations.

These improvements would increase the real-world applicability of the system in various research and conservation scenarios.

# Bibliography


[1] Sara Beery, Dan Morris, and Siyu Yang. *Efficient Pipeline for Camera Trap Image Review*. http://github.com/agentmorris/MegaDetector. arXiv preprint, 1907.06772. 2019. URL: https://arxiv.org/abs/1907.06772.

[2] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[3] Inc Docker et al. "Docker". In: *Web resource: https://www.docker.com/what-docker* (2020).

[4] Miguel Grinberg. *Flask web development*. " O'Reilly Media, Inc.", 2018.

[5] Kinnary Jangla. "Docker compose". In: *Accelerating Development Velocity Using Docker: Docker Across Microservices*. Springer, 2018, pp. 77–98.

[6] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. *Ultralytics YOLO*. https://github.com/ultralytics/ultralytics. Version 8.0.0. Ultralytics, Jan. 2023. URL: https://ultralytics.com.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[8] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (2002), pp. 2278–2324.

[9] Inc. MinIO. *MinIO: High Performance Object Storage*. https://min.io/. 2015.

[10] Margarita Mulero-Pázmány et al. "Addressing significant challenges for animal detection in camera trap images: a novel deep learning-based approach". In: *Scientific Reports* 15.1 (2025), pp. 1–18.

[11] Behandelt PostgreSQL. "PostgreSQL". In: *Web resource: http://www.PostgreSQL.org/about* (1996).

[12] Why Python. “Python”. In: *Python programming language* 24 (2021).

[13] Sovit Ranjan Rath. *Faster R-CNN PyTorch Training Pipeline*. https://github.com/sovit-123/fasterrcnn-pytorch-training-pipeline/tree/main. 2024.

[14] Jeff Reback et al. “pandas-dev/pandas: Pandas”. In: *Zenodo* (2020).

[15] Shaoqing Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (June 2017).

[16] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.

[17] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536.

[18] Salvatore Sanfilippo and Redis Contributors. *Redis: In-Memory Data Structure Store*. https://redis.io/. 2009.

[19] Ask Solem and Contributors. *Celery: Distributed Task Queue*. https://docs.celeryq.dev/. 2009.

[20] Alexandra Swanson et al. “Snapshot Serengeti, high-frequency annotated camera trap images of 40 mammalian species in an African savanna”. In: *Scientific data* 2.1 (2015), pp. 1–14.

# A Installation Manual

## A.1 Requirements

- Docker and Docker Compose.
- 8 GBs of RAM.
- MinIO credentials.

## A.2 Running the Application

Clone the repository and navigate to the `src` directory:

```
git clone https://github.com/ziriraha/snieves-donana-benchmark.git
```

Create a `.env` file with the following variables (see `example.env`):

- `MINIO_URL`
- `MINIO_ACCESS_KEY`
- `MINIO_SECRET_KEY`
- `MINIO_BUCKET`

To run the app, execute the following command in the `src` directory with Docker and Docker Compose installed:

```
docker compose up --build -d
```

Optionally, you can use `--scale celery=3` to scale Celery to use 3 workers, which can help with performance when processing large datasets. The `config.toml` file can also be modified to add more workers for the Flask app. These actions will consume more resources.

To populate the database, open a shell in the Flask app container and run the `init-db` command:

```
docker compose exec -it app /bin/bash
flask init-db
```

This will run the data importation task. It may take up to 5 minutes for the data to appear in the database.

For generating the train, val, and test dataset zip files:

```
flask download-datasets
```

This will start a Celery task to download the datasets. To view the progress, check the Celery logs. This process may take several hours to complete and requires approximately 200 GB of storage.

In case you need to delete custom datasets:

```
flask delete-custom-datasets
```

Once the app is running, you can access it at `http://localhost:4000/`.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA