2022

C964: Capstone
Bird Classification Application

Kathleen De Ridder

Student #[redacted]

Western Governors University

11/7/2022

Table of Contents

Section A

Letter of Transmittal

November 7th, 2022

Kathleen De Ridder
[redacted], CO, [redacted]

[redacted]
OR, [redacted]

Dear [redacted]

Subject: Web Application proposal for bird identification.

Many people enjoy watching birds in their backyards, but they don't always know which birds they are looking at, or what the best way is to make their backyards more inviting to them. [redacted] is always looking for more ways to engage the general public in the protection and preservation of nature in their surroundings. The first step in this process is to educate people about the birds they share their environment with.

Creating an easy-to-use, easy-to-access tool that allows people to quickly and reliably identify the species of birds in their environment will engage the people with nature around them and can spark an interest in making their backyards more hospital for them. By making nature more recognizable, they will feel more involved and connected to it.

To this end, we propose a web application, accessible through their web browser, that allows a user to upload a picture they have taken of a bird they have seen. The machine learning algorithm powering the application will be able to accurately recognize which bird is depicted in the image. Once the user knows who is visiting their backyards, they can easily find the right ways to make their yards a better place for the birds, by offering them appropriate bird seeds or providing the right-sized birdhouses.

The application we propose will cost an estimated $34,320 to develop. There will not be any need to acquire new hardware for this project. My experience developing multiple web application projects, and several machine learning solutions, qualifies me to successfully complete this project.

If you have any further questions about this proposal, please do not hesitate to contact me.

Sincerely,

Kathleen De Ridder
[redacted]

Project Recommendation

1. Problem Summary

[redacted] is a non-profit organization that aims to increase awareness and involvement of the general public in the preservation of nature found in the public's own neighborhoods. In urban or suburban settings, it is often easy for people to miss or ignore the fauna and flora that are integrated into their environment and neighborhood. [redacted] wants to bring attention to these pockets of nature that are found all around us, in an effort to inspire people to preserve these species and their habitats and to coexist with nature around them.

[redacted] wants to create a tool that allows anybody to identify the birds they see around them, simply by uploading a photo to a web application. By creating an easy-to-use application, the user can get more informed about what lives around their homes, encouraging them to make their backyards more hospitable for birds around them.

2. Application Benefits

By offering users a user-friendly web application, anyone who is interested in the bird population around them can easily get more informed about what species of birds live in their own backyards, without having to manually identify the birds they see. The application will swiftly and accurately predict which bird the user photographed, allowing them to research how to make their environment more welcoming to these birds, for example by hanging out bird feeders with appropriate bird seeds, or by installing birdhouses with the right-sized holes. This awareness of which bird species live in their own neighborhoods can spark an interest in nature around them, and a desire to preserve the species and their habitats.

3. Application Description

The proposed data solution will be a web-based application, hosted on Heroku. The user will be able to access this application in their web browser. Heroku is a Platform as a Service (PaaS) cloud service that allows easy deployment of applications (Heroku, 2022).

The code for the application will be written in python 3, using Jupyter notebooks, and will be visualized with the Voila extension for Jupyter notebooks. The notebooks will use several libraries such as pandas, NumPy, matplotlib, and TensorFlow.

The application will be powered by the convolutional neural network (CNN) called MobileNet V3. This model has been pre-trained on the very large (1.4M) ImageNet dataset (ILSVRC-2012-CLS (Russakovsky et al., 2015)) to be able to recognize images of all kinds. Other CNNs (such as ResNet and EfficientNet) will be explored and compared to MobileNet in terms of accuracy, precision, recall, and size, to verify that this model is the best solution for the application.

Through transfer learning, we will train this model on the bird dataset, so that it will be able to accurately (> 90%) classify birds into 450 different species. We will utilize three different notebooks. The first one will be used to train the model, using the TensorFlow, TensorFlow-hub, and Keras libraries. The

second notebook will create the data visualizations, relying heavily on the matplotlib library. The model and the visualizations will be combined in a third notebook that creates the user interface, using Voila.

The dataset we will use for the training is a public dataset from Kaggle (Piosenka, 2022), containing 75,000 images of birds spread over 450 species and split into a training set, a testing set, and a validation set.

Then the web application will use the trained model to accurately predict the bird in the picture uploaded by the user. The user will be presented with the top 3 predictions and their probabilities.

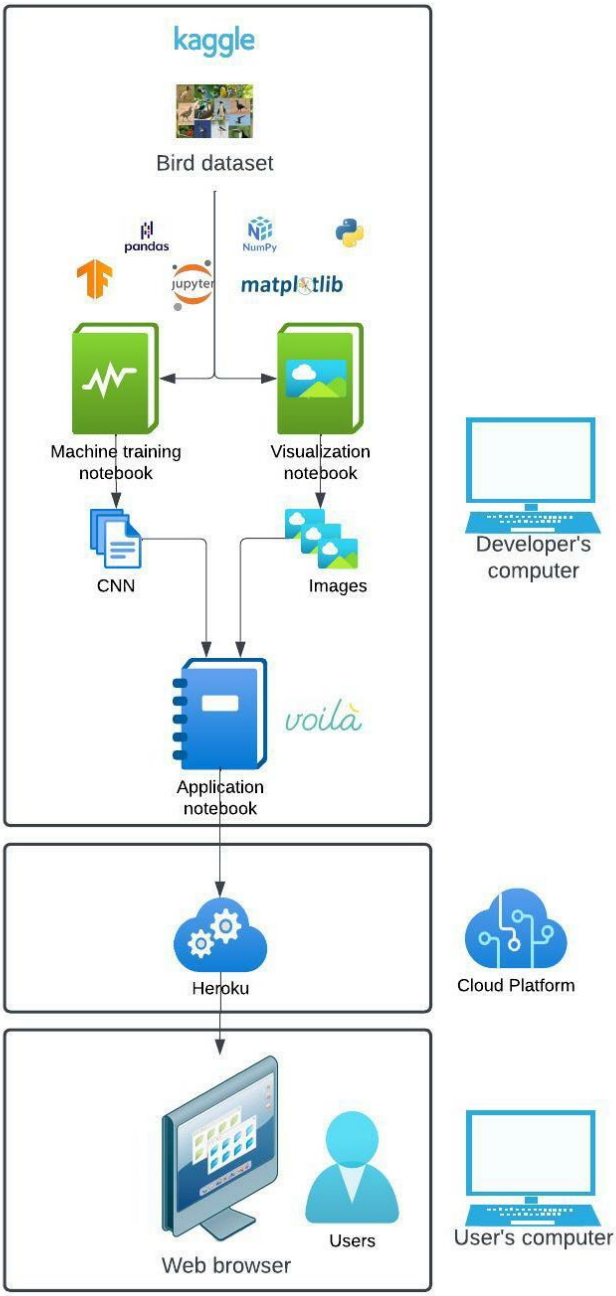


Figure 1 Overview of application.

4. Data Description

The dataset we will use for training the model is a public domain dataset taken from Kaggle (Piosenka, 2022). It is a large dataset (2GB) that contains 70,626 training images (between 130 and 248 images per species), 2250 test images (5 images per species), and 2250 validation images (5 images per species). Since it has at least 100 images per species to train on, we are confident that the model will be able to accurately recognize all bird species.

The images are stored in a directory that contains subdirectories to hold training, testing, and validation images separately. Each subdirectory then contains 450 folders, each named for the species they contain.

The dataset also comes with a csv file containing the following columns:

- class id
- filepaths
- labels
- scientific label
- data set

The label column is the dependent variable, the images are the independent variable.

The dataset was collected by searching the internet for images by species name. Google and Bing tend to skew towards higher ratios of male birds in their search results, and as a result the dataset also tends to be skewed towards male specimens (as high as 80%). This can be significant since many birds have high sexual dimorphism (distinct differences in size, morphology, or coloration between males and females of the same species). An example of this is the Mallard duck, where the males and females have completely different colors. Thus, it is to be expected that the model will do better at classifying males than females, especially in species that have high sexual dimorphism. Because the validation and test images also have the same imbalance, the accuracy of the model might not suffer too much when based on those sets. The real-life accuracy experienced by the users, where this imbalance is of course not present, however, might be lower.

All images have the shape 224 x 224 x 3 (width, height, color channels) and are in jpeg format. The images have been cropped by the author of the dataset to maximize the percentage of the image taken by the actual bird (a minimum of 50% of the pixels in the images are taken by the bird, as opposed to small birds in large backgrounds). This is beneficial for training the model, as it is shown more of the actual bird. However, because the images are of such good quality (close-up birds, cropped to keep only the bird, only one bird per image) users might experience a lower accuracy with their own pictures that might be taken from further away, might be blurrier and are most likely not cropped to keep only the bird.

5. Objective and Hypotheses

Our project hypothesizes that we can train a CNN model so that it will accurately (> 90%) classify bird images into 450 different species.

The objective is to provide an easy-to-use web application that allows a user to upload a picture containing a bird and get a prediction of which species the bird most likely belongs to. This can increase interest and awareness of nature in the user's environment. We also want to present the user with a descriptive analysis of the dataset, in the form of two static visuals and one interactive visual.

6. Methodology

We will use the waterfall methodology for this project. The scope of the project is relatively small and well-defined at the start of the project. Changes in requirements or objectives are unlikely and probably small if they do occur.

The following steps will be taken in this project:

1. **Requirements**: Gather and list all requirements for the entire project.
2. **Design**: Design of the data solution.
3. **Implementation**: Writing the code for the application.
4. **Verification/Testing:** Test the application to assure all requirements are met and to minimize bugs.
5. **Deployment**: Make the application available to users.
6. **Maintenance**: Monitor logs and deploy updates to fix bugs or defects.

7. Funding Requirements

	Cost / Hour	Total hours	Total cost
Planning & Design	$ 120	60	$ 7,200
Development	$ 120	200	$ 24,000
Office space	$ 12	260	$ 3,120
Cloud service (Heroku)*	$ 0		$ 0
Total			**$ 34,320**

Table 1 Funding requirements.

8. Stakeholders' Impact

This application will increase awareness about birds and spark the interest of the users in the bird population around them. This will benefit both the user, by bringing them enjoyment and educating them, and [redacted] by helping it in its efforts to protect and conserve nature in (sub-)urban settings and to raise awareness for fauna around humans.

9. Data Precautions

The dataset that is being used is in the public domain and contains only images of birds. No humans are depicted in any of the images. The dataset does not contain any personal or sensitive information. None of the images contain any inappropriate or cruel behavior towards animals, and they only depict birds in their natural habitat.

10. Developer's Expertise

I have the required expertise to successfully complete the project. I have two years of experience programming in python and have previously completed web application projects, some of which also used TensorFlow to train machine learning models.

Section B

1. Problem Statement

People generally feel more involved and connected with things around them that they can recognize or know something about. Creating a web-based application that allows users to quickly and easily classify birds in their immediate environment will bring those users closer to nature and spark a desire to preserve the habitats and species that live around them.

This tool needs to be readily available, easy to use, relatively fast, and accurate for users to use it and to keep using it. We, therefore, propose to make a web-based application that is available through the users' web browsers without the need to install anything. Identifying a bird should be as easy as uploading a picture from their computer. The machine learning algorithm that powers the application should be at least 90% accurate in predicting the species in the picture.

2. Customer Summary

This application is aimed to be used by the general public, specifically people interested in learning about the diversity of birds in their own neighborhoods. No special skills or knowledge are required to use the application. No installation is necessary, the user can simply use their web browser to access the application. While the application can be used on a mobile environment, it will mostly be optimized for a laptop or desktop computer. By keeping the application simple and easy to access, the threshold for users to educate themselves on the diversity of birds around them will be lowered.

3. Existing System Analysis

Currently, [redacted] website features pictures and information about birds commonly found in (sub)urban settings, so that interested users can try to identify the birds they might have spotted in their neighborhood. This approach is not optimal, because subtle differences between two bird species might not be very obvious to the untrained eye, and differentiating between bird species can be hard, especially based on the one or two pictures that are shown on the website. It also requires a lot of searching for the right bird among the many species that are shown on the website. This can cause some users to lose interest.

The machine learning algorithm in the solution we propose will be able to classify a bird in a picture automatically, without the user needing to search for potential matches. Because the algorithm is trained on a very large set of images, featuring over a hundred images per species, it will be able to differentiate between two species of birds that might look very similar to casual viewers. The user can find additional information about the species of bird the application has identified in their picture, without having to search and compare manually, making the experience much more convenient and easier for the user.

4. Data

The dataset that will be used is a publicly available dataset from Kaggle (Piosenka, 2022). It is available at this web address [redacted]

The dataset contains over 75,000 images of birds. They are already divided into training (70,626), testing (2250), and validation (2250) subsets. The dataset contains 450 species of birds. Both the training and validation subsets contain 5 images per species, the training set contains between 130 and 248 images per species. Because there are so many pictures available in this dataset, we expect the model to have an accuracy of over 90%.

The dataset was collected by searching the internet for images by species name. Google and Bing tend to skew towards higher ratios of male birds in their search results, and as a result the dataset also tends to be skewed towards male specimens (as high as 80%). This can be a significant difference since many birds have high sexual dimorphism (distinct differences in size, morphology, or coloration between males and females of the same species). An example of this is the Mallard duck, where the males and females have completely different colors. Thus, it is to be expected that the model will do better at classifying males than females, especially in species that have high sexual dimorphism. Because the validation and test images also have the same imbalance, the accuracy of the model might not suffer too much when based on those sets. The real-life accuracy, experienced by the users, where this imbalance is of course not present, might therefore be lower.

All images have the shape 224 x 224 x 3 (width, height, color channels) and are in jpeg format. The images have been cropped by the author of the dataset to maximize the percentage of the image taken by the actual bird (a minimum of 50% of the pixels in the images are taken by the bird, as opposed to small birds in large backgrounds). This is beneficial for training the model, as it is shown more of the actual bird. However, because the images are of such good quality (close-up birds, cropped to keep only the bird, only one bird per image) users might experience a lower accuracy with their own pictures that might be taken from further away, might be blurrier and are most likely not cropped to keep only the bird.

The images are stored in a directory that contains subdirectories to hold training, testing, and validation images separately. Each subdirectory then contains 450 folders, each named for the species they contain.

The dataset also comes with a csv file containing the following columns:

- **class id:** Unique id per species (0 – 449)
- **filepaths**: Relative path to the image. Example: “train/ABBOTTS BABBLER/001.jpg” where:
  - “train” refers to the training subdirectory,
  - “ABBOTS BABBLER” refers to the species subdirectory and
  - “001” refers to the first image of this species.
- **labels**: Bird species in the image.
- **scientific label**: The name of the species in the standard binomial nomenclature, where the first part of the name refers to the genus and the second part refers to the species. Example: Malacocincla *abbotti*
- **data set**: This refers to which subset of the dataset (train/test/validation) the image belongs to.

The label column is the dependent variable, and the images (filepaths) are the independent variable.

From the labels column, we can extract a list of unique labels that our model will be able to recognize, by taking the unique values.

Each label will also be converted to Boolean form, in order to match the output of the model. This converts each label to an array of 450 values, all of them 'False' except the value at the place of the label this array represents. For example, the label "ABBOTS BABBLER", the first label, will look like: ['True', 'False', 'False',…]. The model will output a similar array of 450 values, all between 0 and 1, and all adding up to 1, where each value represents the probability that the bird in the picture belongs to each of the species. If we already have our labels in Boolean form, we can simply check which label has a 'True' at the index with the highest probability, to get the predicted label.

The scientific label is generally in the standard binomial nomenclature form or the "Latin name" of the species. This refers to the naming convention where species have a two-part name: the first part refers to their order and the second part to their genus. For example, in Malacocincla *abbotti,* Malacocincla refers to the genus, and *abbotti* to the specific species.

From this column, we will first extract the genus of the bird by splitting the scientific name into the genus and the species. Cross-referencing the genus with a separate list of bird families and the genera and species that belong to those families, found [redacted] (Boyd, 2019), will allow me to assign a family to each bird species. The dataset spans 450 species, but those can be reduced to 136 families. Grouping by family rather than by species allows for a more uncluttered view of the data, while still keeping closely related birds together.

Not all the names in the scientific label column follow the two-part naming convention. For example, the antbirds, Thamnophilidae, are only referred to by their family name, and no specific genus or species is given. In those cases, we will refer to both their family, genus, and species with the same name.

In a handful of cases, the genus name given in the bird dataset does not correspond with any genus name in the family list. This can have many reasons, for example, small differences in spelling or conflicting classifications of birds, based on different classification systems used. Biologists continually make changes in classifications, based on new evidence. We can choose to omit these special cases from our data, or we can manually adjust them. Since there are not too many of these mismatches (<20), we will manually check the 'correct' family and assign it to the bird. This way there will not be any data missing.

TensorFlow is optimized to work best when the data is provided to it in batches. We will therefore convert the images to tensors and then group them in batches of 32. Any results from the model will also be in batch form and will need to be converted back before they can be read.

5. Project Methodology

The waterfall methodology encompasses the following phases:

1. **Requirements**:

   We will meet with [redacted] and gather all the requirements for the project. These requirements should encompass all that the client likes to see in the finished application.

2. **Design**:

Based on the requirements, the overall design of the application will be determined. Decisions such as the layout, programming language, and hosting service will be determined in this phase.

3. **Implementation**:

Writing the application code.

   a. The data will be collected.
   b. The environment will be set up using miniconda
   c. Data analysis
   d. Creating visual representations of the data
   e. Data preprocessing
   f. Creating the model
   g. Training the model
   h. Evaluating the model
   i. Comparing the models and if needed finetuning steps f through h until acceptance goals (>90% accuracy) are met
   j. Creating the user interface
   k. Testing
   l. Deployment to Heroku
   m. User guide creation

4. **Verification/Testing:**
   a. Unit and integration testing
   b. Acceptance testing
5. **Deployment**:
   The application will be uploaded to Heroku so that it becomes publicly available.
6. **Maintenance**:
   Heroku provides logs in the developer's dashboard that can be used to monitor the activity of the application and identify any possible problems with the application.

6. Project Outcomes

The outcome of the project is the deployment of a fully functional web application, hosted on Heroku, that allows the user to upload their own picture and receive a predicted label in return. The user will also see a bar graph showing the top 3 predicted labels. The web application also allows the user to view two static data analysis visuals: a bar chart showing the number of images in the dataset for each bird species, and a scatter plot showing the number of images in the dataset per bird family. The user will also be able to view an interactive visual, which allows the user to choose a bird family and see the average RGB values for each image in that family.

Project Deliverables:

- Three **jupyter notebooks:**
  - One for training the model
  - One for creating the data visualizations

    - One for the user interface (included in the application)
- **Three trained models** from which the most suitable will be chosen.

Product Deliverables:

- A fully trained and optimized **model,** capable of classifying birds with a minimum of 90% accuracy.
- Three **csv files**:
    - **Unique_labels.csv** holds the list of 450 unique bird species that the model will be able to differentiate.
    - **Unique_families.csv** holds the list of the unique 136 families that our bird species can be grouped in.
    - **Bird_rgb.csv** holds a column of average RGB values for each image, and a column of which family the image belongs to.
- A jupyter notebook that creates **the user interface**
- Two **static visuals**:
    - Bar chart of the number of images per species
    - Scatter plot of the number of images per family
- **Interactive visual**:
    - 3D scatter plot of average RGB values per image for a chosen family
- An easy-to-use user interface hosted on **Heroku**, that incorporates both the classifying functionality as well as two static data analysis visuals and one interactive data analysis visual.
- Files needed for deployment on Heroku:
    - **requirements**: lists the required libraries needed to build the environment on Heroku to run the application, as well as their version numbers (to avoid dependency issues).
    - **runtime**: lists the programming language used (python 3.10.8)
    - **procfile**: tells Heroku what kind of application it is (web) and how it should be launched. We will specify here that we want Heroku to automatically open the notebook with the voila library, so we only see our dashboard.
- A **user guide**

7. Implementation Plan

a. Data gathering:
    i. The bird dataset will be downloaded from Kaggle (Piosenka, 2022): [redacted]
    ii. The bird-species-list will be downloaded from the Taxonomy in Flux website (Boyd, 2019): [redacted]
b. The environment will be set up using miniconda.
c. Data analysis.
d. Creating visual representations of the data:
    i. Bar chart of the number of images per species.
    ii. Scatter plot of the number of images per family.
    iii. 3D Scatter plot of average RGB values per image for a chosen family.

e. Data preprocessing:
    i. Converting labels to Boolean format.
    ii. Making separate lists of training, testing, and validation images (already split in the dataset) and their corresponding labels.
    iii. Grouping images in batches.
    iv. Converting images to tensors.
f. Creating the model using the URL for the MobileNetV3 pre-trained model (Howard et al., 2019): https://tfhub.dev/google/imagenet/mobilenet_v3_large_075_224/classification/5
g. Training the model using the training subset.
h. Evaluating the model.
i. Comparing the models and if needed finetuning steps f through h until acceptance goals (>90% accuracy) are met.
j. Creating the user interface using the voila library and jupyter notebooks.
k. Testing:
    i. Unit testing on individual components.
    ii. Integration testing.
    iii. Acceptance testing will be done by the team of [redacted]
l. Deployment to Heroku.
m. Creating the user guide.

8. Evaluation Plan

We will evaluate and compare the different models (MobileNet, ResNet, and EfficientNet) using three metrics:

- **Accuracy** is the ratio of correct predictions over the total number of predictions. An accuracy of 90% means that of the 10 pictures we feed into the model, we expect 9 of them to be labeled correctly. This metric takes an average over all the classes, so smaller classes might be underrepresented in this metric. We want our model to have an accuracy of at least 90%.
- **Precision** is the ratio of true positives over all positives. It measures how accurate the model is in classifying a bird as a certain species. For example: if we let the model predict 10 images, and it correctly identifies 4 of those as swans, but also (falsely) identifies 3 other birds as swans, the precision will only be 57%.
- **Recall** is the ratio of true positives over the total number of positives. Unlike precision, it does not take false positives into account. For example: if our set of 10 images contains 4 swan images, but the model only correctly classifies 3 of those as swans, then we have a recall of 75%, regardless of how many other birds are also classified as swans.

The model should have an **accuracy of at least 90%,** measured using the testing subset. We will also look at the other metrics and compare the size of the trained models. Heroku has a limited amount of memory that can be used by the application, so we need to make sure the model is lightweight.

We will perform unit testing and integration testing to assure all individual components work as intended and work together as intended. The customer ([redacted] will do acceptance testing to assure the product meets their needs.

9. Resources and Costs

Programming Environment:

- Development of the application will be done on the developer's computer, which runs on Windows 10. It has an NVIDIA Geforce GTX 1080 GPU. It has an Intel i7 4 Core CPU with 32.0 GB RAM.
- The coding will be done in Jupyter notebooks, using python 3.10.
- Miniconda will be used to create two separate environments:
  A. The notebook that does the model training will run in an environment that includes the GPU-enabled version of TensorFlow. This allows us to utilize the GPU to train the model. Because the dataset contains so many training images (> 75k), a GPU is needed to train the model in a reasonable time. Because of dependency conflicts, this environment cannot also contain matplotlib, the library we will need to create the visualizations. We will use a separate environment and notebook for this.

     This environment will include the following major libraries:
     - Tensorflow (full GPU version)
     - Tensorflow Hub
     - NumPy
     - Pandas
     - IPython

  B. The notebooks for the visualizations and the user interface will run in a separate environment that has the CPU-only version of TensorFlow installed. This version does not allow the use of the GPU, but it can be combined with matplotlib, and it requires a lot less memory. This is important especially for the user interface because Heroku has a limited available memory quota, and the environment takes up the majority of the memory needed by the application.

     This environment will include the following major libraries:
     - Tensorflow-CPU
     - Tensorflow hub
     - NumPy
     - Pandas
     - Matplotlib
     - Ipywidgets
     - PIL
     - IPython
     - Voila
- The finished application will be hosted online on Heroku.com. The user interface notebook will contain the GUI that is visualized using the voila library to provide the user with a dashboard. The requirements document will contain all the required libraries needed for the Heroku platform to build the correct environment.

Environment costs:

The environment will not require the purchase of any new materials or tools. The libraries and tools we will use are free and open-source. The rent for the office space where the application will be developed is $12 per hour. For a total expected development time of 260 hours, this comes to $3,120.

Hosting on Heroku is currently free, but this cost might increase if more bandwidth or memory is needed in the future, or if the terms of service change.

Human resources costs:

	Cost / Hour	Total hours	Total cost
Planning & Design	$ 120	60	$ 7,200
Development	$ 120	200	$ 24,000

Table 2 Human resources costs.

Total overview:

	Cost / Hour	Total hours	Total cost
Planning & Design	$ 120	60	$ 7,200
Development	$ 120	200	$ 24,000
Office space	$ 12	260	$ 3,120
Cloud service (Heroku)	$ 0		$ 0
Total			**$ 34,320**

Table 3 Total costs overview.

10. Timeline and Milestones

Since all the resources (developer's salary and office rental) depend on the number of hours spend on each task, we will list them as the time (in hours) needed to complete the task.

Milestone	Start and end dates	Duration (days)	Dependencies	Resources (hours)
Data gathering	10/20/22	1		4
Environment setup	10/20/22	1		4
Data analysis	10/20/22 – 10/21/22	2	Data gathering, environment setup	16
Visuals Creation	10/24/22 – 10/28/22	5	Data analysis	40
Data preprocessing	10/31/22 – 11/1/22	2	Data gathering, environment setup	8
Model creation	11/2/22	1	Data preprocessing	4
Model training	11/2/22	1	Model creation	4

Model evaluation and comparison	11/3/22 – 11/4/22	2	Model training	8
User interface creation	11/7/22 – 11/11/22	5	Environment setup	40
Testing	11/14/22 – 11/18/22	5	Model evaluation, visuals creation, user interface creation	40
Deployment to Heroku	11/21/22 – 11/21/22	1	Model evaluation, visuals creation, user interface creation, testing	8
User guide creation	11/22/22 – 11/24/22	3	Deployment	24

Table 4 Project milestones.

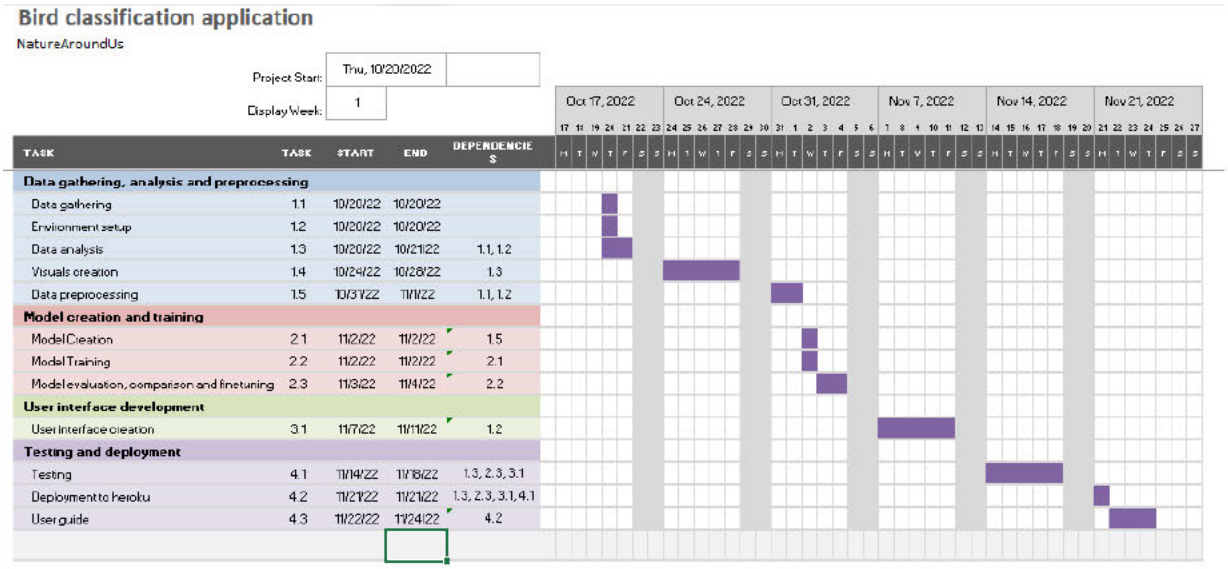


Figure 2 Project Gantt chart.

Section D

1. Project Purpose

[redacted] is a non-profit organization that aims to increase awareness and involvement of the general public in the preservation of nature found in the public's neighborhoods. To bring people closer to the bird population living among them, a web application was developed that allows the user to upload a picture they took of a bird. The application is capable of recognizing which bird is depicted in the image and displays the species name to the user, as well as the top 3 predicted species and the probabilities that the bird in the image belongs to each of those species. The underlying machine learning model has an accuracy of 95% and the application itself is intuitive and easy to use.

The users can also explore some of the dataset characteristics through the visuals integrated into the application, such as the bar chart showing the number of images of each bird species in the dataset, the scatter plot showing the number of images of each bird family in the dataset and the 3D interactive scatter plot that shows the average RGB values for each image in a chosen bird family.

Having an easy way to learn more about these animals increases awareness, interest, and goodwill in the public which can lead to a better coexistence between humans and animals.

2. Datasets

The main dataset is a publicly available bird dataset found on Kaggle (Piosenka, 2022) at this URL: [redacted]

The dataset was already very organized and clean, so very little processing was needed to make it useful for our application. The set contains a total of 75,126 images, divided into 70,626 training images, 2250 testing images, and 2250 validation images, already separated into three subdirectories. The dataset contains 450 species of birds. Both the training and validation subsets contain 5 images per species, the training set contains between 130 and 248 images per species.

Check number of images per species

```
In [21]: # Number of images in training set per species
         birds.loc[birds['data set'] == "train"]["labels"].value_counts()

Out[21]: HOUSE FINCH                248
         D-ARNAUDS BARBET          233
         OVENBIRD                  233
         SWINHOES PHEASANT         217
         WOOD DUCK                 214
                                  ...
         GO AWAY BIRD              131
         BLACK FRANCOLIN           131
         RED TAILED THRUSH         130
         PATAGONIAN SIERRA FINCH   130
         SNOWY PLOVER              130
         Name: labels, Length: 450, dtype: int64
```

Figure 3 Number of images per species.

All images had already been cropped to maximize the bird in the picture and they had all been resized to the shape 224 x 224 x 3 (width, height, color channels) and stored in jpeg format. Since birds don't change their appearances, the time or year in which the images were taken/collected is largely irrelevant. The dataset contains images of birds occurring all over the world, so the place the picture was taken is also irrelevant.

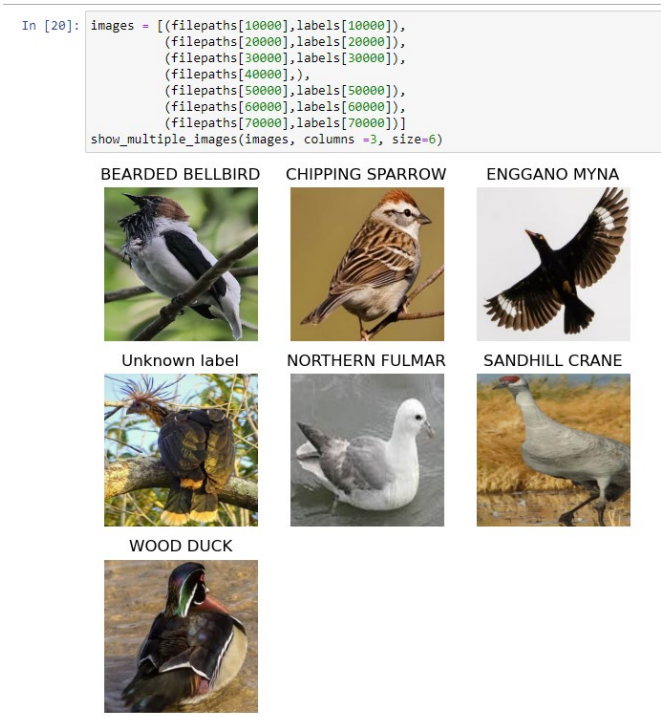


Figure 4 Visualization of a handful of randomly selected images.

The accompanying csv file lists the following features for each image:

- **class id:** Unique id per species (0 – 449)
- **filepaths**: Relative path to the image.
- **labels**: Bird species in the image.
- **scientific label**: The name of the species in the standard binomial nomenclature.
- **data set**: train/test/validation

	A	B	C	D	E
1	class id	filepaths	labels	scientific label	data set
164	0	train/ABBOTTS BABBLER/163.jpg	ABBOTTS BABBLER	Malacocincla abbotti	train
165	0	train/ABBOTTS BABBLER/164.jpg	ABBOTTS BABBLER	Malacocincla abbotti	train
166	0	train/ABBOTTS BABBLER/165.jpg	ABBOTTS BABBLER	Malacocincla abbotti	train
167	0	train/ABBOTTS BABBLER/166.jpg	ABBOTTS BABBLER	Malacocincla abbotti	train
168	1	train/ABBOTTS BOOBY/001.jpg	ABBOTTS BOOBY	Papasula abbotti	train
169	1	train/ABBOTTS BOOBY/002.jpg	ABBOTTS BOOBY	Papasula abbotti	train
170	1	train/ABBOTTS BOOBY/003.jpg	ABBOTTS BOOBY	Papasula abbotti	train
171	1	train/ABBOTTS BOOBY/004.jpg	ABBOTTS BOOBY	Papasula abbotti	train
172	1	train/ABBOTTS BOOBY/005.jpg	ABBOTTS BOOBY	Papasula abbotti	train
173	1	train/ABBOTTS BOOBY/006.jpg	ABBOTTS BOOBY	Papasula abbotti	train
174	1	train/ABBOTTS BOOBY/007.jpg	ABBOTTS BOOBY	Papasula abbotti	train

Figure 5 Example of the raw dataset csv file.

From this dataset, we extracted and saved a list of unique bird labels, which is the 450 bird species that our model can recognize. Each label is then converted to Boolean form.

Create unique labels array ¶

This array holds all the different species that our model will be able to classify (450 unique bird species).

```
In [15]: unique_labels = np.asarray(birds["labels"].unique())
         unique_labels

Out[15]: array(['ABBOTTS BABBLER', 'ABBOTTS BOOBY', 'ABYSSINIAN GROUND HORNBILL',
                'AFRICAN CROWNED CRANE', 'AFRICAN EMERALD CUCKOO',
                'AFRICAN FIREFINCH', 'AFRICAN OYSTER CATCHER',
                'AFRICAN PIED HORNBILL', 'ALBATROSS', 'ALBERTS TOWHEE',
                'ALEXANDRINE PARAKEET', 'ALPINE CHOUGH', 'ALTAMIRA YELLOWTHROAT',
                'AMERICAN AVOCET', 'AMERICAN BITTERN', 'AMERICAN COOT',
                'AMERICAN FLAMINGO', 'AMERICAN GOLDFINCH', 'AMERICAN KESTREL',
                'AMERICAN PIPIT', 'AMERICAN REDSTART', 'AMERICAN WIGEON',
                'AMETHYST WOODSTAR', 'ANDEAN GOOSE', 'ANDEAN LAPWING',
                'ANDEAN SISKIN', 'ANHINGA', 'ANIANIAU', 'ANNAS HUMMINGBIRD',
                'ANTBIRD', 'ANTILLEAN EUPHONIA', 'APAPANE', 'APOSTLEBIRD',
                'ARARIPE MANAKIN', 'ASHY STORM PETREL', 'ASHY THRUSHBIRD',
                'ASIAN CRESTED IBIS', 'ASIAN DOLLARD BIRD', 'AUCKLAND SHAQ',
                'AUSTRAL CANASTERO', 'AUSTRALASIAN FIGBIRD', 'AVADAVAT',
                'AZARAS SPINETAIL', 'AZURE BREASTED PITTA', 'AZURE JAY',
                'AZURE TANAGER', 'AZURE TIT', 'BAIKAL TEAL', 'BALD EAGLE',
                'BALD IBIS', 'BALI STARLING', 'BALTIMORE ORIOLE', 'BANANAQUIT',
                'BAND TAILED GUAN', 'BANDED BROADBILL', 'BANDED PITA',
                'BANDED STILT', 'BAR-TAILED GODWIT', 'BARN OWL', 'BARN SWALLOW',
```

Figure 6 Unique labels array.

To add the family name to each bird species, we retrieve a list of all bird families and the genera that they contain from the Taxonomy in Flux website (Boyd, 2019): [redacted]

	A	B	C	D	E	F
1	Number	Family	Family Name	Genus	Species	English Name
2	1	Struthionidae	Ostriches	Struthio	camelus	Common Ostrich
3	2	Struthionidae	Ostriches	Struthio	molybdophanes	Somali Ostrich
4	3	Rheidae	Rheas	Rhea	americana	Greater Rhea
5	4	Rheidae	Rheas	Rhea	pennata	Lesser Rhea
6	5	Casuariidae	Emus & Cassowaries	Dromaius	novaehollandiae	Emu
7	6	Casuariidae	Emus & Cassowaries	Casuarius	casuarius	Southern Cassowary
8	7	Casuariidae	Emus & Cassowaries	Casuarius	bennetti	Dwarf Cassowary
9	8	Casuariidae	Emus & Cassowaries	Casuarius	unappendiculatus	Northern Cassowary
10	9	Apterygidae	Kiwis	Apteryx	owenii	Little Spotted Kiwi
11	10	Apterygidae	Kiwis	Apteryx	haastii	Great Spotted Kiwi
12	11	Apterygidae	Kiwis	Apteryx	australis	Southern Brown Kiwi
13	12	Apterygidae	Kiwis	Apteryx	mantelli	North Island Brown Kiwi
14	13	Apterygidae	Kiwis	Apteryx	rowi	Okarito Kiwi
15	14	Tinamidae	Tinamous	Eudromia	elegans	Elegant Crested-Tinamou
16	15	Tinamidae	Tinamous	Eudromia	formosa	Quebracho Crested-Tinamou
17	16	Tinamidae	Tinamous	Tinamotis	pentlandii	Puna Tinamou
18	17	Tinamidae	Tinamous	Tinamotis	ingoufi	Patagonian Tinamou
19	18	Tinamidae	Tinamous	Rhynchotus	rufescens	Red-winged Tinamou

Figure 7 Raw birds family list.

After reading this csv file into the notebook, we can clean it up a little by removing unnecessary columns and dropping duplicate rows.

```
In [27]: # Remove unnecessary columns
bird_species_list = bird_species_list.drop(['English Name', 'Species', 'Number'], axis=1)
# Drop duplicate rows
bird_species_list = bird_species_list.drop_duplicates()
# Check cleaned up list
bird_species_list.head()
```

Out[27]:

	Family	Family Name	Genus
0	Struthionidae	Ostriches	Struthio
2	Rheidae	Rheas	Rhea
4	Casuariidae	Emus & Cassowaries	Dromaius
5	Casuariidae	Emus & Cassowaries	Casuarius
8	Apterygidae	Kiwis	Apteryx

Figure 8 Cleaned-up birds' family list.

Before we can add the family name to the birds dataframe, we need to split the scientific name into genera and species. We only keep the genus.

```
In [28]: # Make a duplicate birds dataframe we can alter without changing the original bird dataframe
birds_altered = birds

# Split the scientific name up in genus and species
birds_altered['genus'] = birds_altered['scientific label'].str.split()

# Keep only the genus (first part of the scientific label)
birds_altered['genus'] = birds_altered['genus'].map(lambda x: x[0])

# Check the new dataframe
birds_altered
```

Out[28]:

	class id	filepaths	labels	scientific label	data set	genus
0	0	train/ABBOTTS BABBLER/001.jpg	ABBOTTS BABBLER	Malacocincla abbotti	train	Malacocincla
1	0	train/ABBOTTS BABBLER/002.jpg	ABBOTTS BABBLER	Malacocincla abbotti	train	Malacocincla
2	0	train/ABBOTTS BABBLER/003.jpg	ABBOTTS BABBLER	Malacocincla abbotti	train	Malacocincla
3	0	train/ABBOTTS BABBLER/004.jpg	ABBOTTS BABBLER	Malacocincla abbotti	train	Malacocincla
4	0	train/ABBOTTS BABBLER/005.jpg	ABBOTTS BABBLER	Malacocincla abbotti	train	Malacocincla
...	...	...	...	...	...	...
75121	449	valid/YELLOW HEADED BLACKBIRD/1.jpg	YELLOW HEADED BLACKBIRD	Xanthocephalus	valid	Xanthocephalus
75122	449	valid/YELLOW HEADED BLACKBIRD/2.jpg	YELLOW HEADED BLACKBIRD	Xanthocephalus	valid	Xanthocephalus
75123	449	valid/YELLOW HEADED BLACKBIRD/3.jpg	YELLOW HEADED BLACKBIRD	Xanthocephalus	valid	Xanthocephalus
75124	449	valid/YELLOW HEADED BLACKBIRD/4.jpg	YELLOW HEADED BLACKBIRD	Xanthocephalus	valid	Xanthocephalus
75125	449	valid/YELLOW HEADED BLACKBIRD/5.jpg	YELLOW HEADED BLACKBIRD	Xanthocephalus	valid	Xanthocephalus

75126 rows × 6 columns

Figure 9 Creation of genus column.

Then we can join the bird family list with the birds dataframe so that the birds dataframe gains two extra columns: one with the scientific family name and one with the common family name.

```
In [29]: # Join the two dataframes to add the family name based on the genus
         joined_birds = birds_altered.set_index('genus').join(bird_species_list.set_index('Genus'))

         # Sort
         joined_birds.sort_values(by=['class id'])
```

Out[29]:

	class id	filepaths	labels	scientific label	data set	Family	Family Name
Malacocincla	0	train/ABBOTTS BABBLER/031.jpg	ABBOTTS BABBLER	Malacocincla abbotti	train	Pellorneidae	Ground Babblers
Malacocincla	0	train/ABBOTTS BABBLER/114.jpg	ABBOTTS BABBLER	Malacocincla abbotti	train	Pellorneidae	Ground Babblers
Malacocincla	0	train/ABBOTTS BABBLER/115.jpg	ABBOTTS BABBLER	Malacocincla abbotti	train	Pellorneidae	Ground Babblers
Malacocincla	0	train/ABBOTTS BABBLER/116.jpg	ABBOTTS BABBLER	Malacocincla abbotti	train	Pellorneidae	Ground Babblers
Malacocincla	0	train/ABBOTTS BABBLER/117.jpg	ABBOTTS BABBLER	Malacocincla abbotti	train	Pellorneidae	Ground Babblers
...	...	...	...	...	...	...	...
Xanthocephalus	449	train/YELLOW HEADED BLACKBIRD/106.jpg	YELLOW HEADED BLACKBIRD	Xanthocephalus	train	Icteridae	New World Blackbirds
Xanthocephalus	449	train/YELLOW HEADED BLACKBIRD/105.jpg	YELLOW HEADED BLACKBIRD	Xanthocephalus	train	Icteridae	New World Blackbirds
Xanthocephalus	449	train/YELLOW HEADED BLACKBIRD/104.jpg	YELLOW HEADED BLACKBIRD	Xanthocephalus	train	Icteridae	New World Blackbirds
Xanthocephalus	449	train/YELLOW HEADED BLACKBIRD/102.jpg	YELLOW HEADED BLACKBIRD	Xanthocephalus	train	Icteridae	New World Blackbirds
Xanthocephalus	449	train/YELLOW HEADED BLACKBIRD/126.jpg	YELLOW HEADED BLACKBIRD	Xanthocephalus	train	Icteridae	New World Blackbirds

75126 rows × 7 columns

Figure 10 Adding scientific family name and common family name.

At this point, we checked to make sure each species had been assigned its family, but we noticed there were 3274 images for which no family was added. This corresponds to around 20 species.

```
In [30]: # Check if there are any rows that did not get a family name
         joined_birds.where(joined_birds["Family Name"].isna()).count()
```

```
Out[30]: class id           3274
         filepaths          3274
         labels             3274
         scientific label   3274
         data set           3274
         Family                0
         Family Name           0
         dtype: int64
```

There are 3274 images where no family name has been added.

Figure 11 Images without a family name.

This might have happened because of alternative spellings, changes in the classification of the birds, or different classification systems used. We can opt to either remove these birds from the statistics or we can alter the family name to make them match. Since there are not too many instances of this, we chose here to manually adjust the family names where needed, so we do not lose any information.

```
# A small function (replace_family_name() ) will replace the "incorrect" family names with the ones that match the
# bird_species_list so that the family can be added correctly to the dataframe.
# Because there are only a handful of species that need to be altered, we add them manually. This process requires googling the
# species name and adding the correct family name for each missing species.

scientific_names = []
families = []
family_names = []

scientific_names.append("Diomedeidae")
families.append("Diomedeidae")
family_names.append("Albatrosses")

scientific_names.append("Thamnophilidae")
families.append("Thamnophilidae")
family_names.append("Antbirds")
```

...

```
scientific_names.append("Ramphastidae")
families.append("Ramphastidae")
family_names.append("Toucans")

# Because of the unusual name of this bird, the function did not work for this special case, so we replace this one manually
joined_birds.loc[joined_birds["labels"] == "IWI", "Family"] = "Fringillidae"
joined_birds.loc[joined_birds["labels"] == "IWI", "Family Name"] = "Finches"
joined_birds.loc[joined_birds["labels"] == "IWI", "scientific label"] = "Drepanis coccinea"

# Run the function that corrects the names
replace_family_name(scientific_names, families, family_names)
```

Figure 12 Manual adjustment of missing family names.

```
def replace_family_name(scientific_names, families, family_names):
    """
    This function replaces the incorrect family names to match our dataset.
    """
    global joined_birds     # Replaces in place
    for i, bird in enumerate(scientific_names):
        joined_birds.loc[joined_birds["scientific label"] == bird, "Family"] = families[i]
        joined_birds.loc[joined_birds["scientific label"] == bird, "Family Name"] = family_names[i]
```

Figure 13 Function replace_family_name().

The resulting dataframe has the scientific family name and common family name added for each image.

```
In [33]: joined_birds
```

Out[33]:

	class id	filepaths	labels	scientific label	data set	Family	Family Name
Accipiter	321	train/NORTHERN GOSHAWK/001.jpg	NORTHERN GOSHAWK	Accipiter gentilis	train	Accipitridae	Hawks
Accipiter	321	train/NORTHERN GOSHAWK/002.jpg	NORTHERN GOSHAWK	Accipiter gentilis	train	Accipitridae	Hawks
Accipiter	321	train/NORTHERN GOSHAWK/003.jpg	NORTHERN GOSHAWK	Accipiter gentilis	train	Accipitridae	Hawks
Accipiter	321	train/NORTHERN GOSHAWK/004.jpg	NORTHERN GOSHAWK	Accipiter gentilis	train	Accipitridae	Hawks
Accipiter	321	train/NORTHERN GOSHAWK/005.jpg	NORTHERN GOSHAWK	Accipiter gentilis	train	Accipitridae	Hawks
...	...	...	...	...	...	...	...
Zosterops	302	valid/MALAGASY WHITE EYE/1.jpg	MALAGASY WHITE EYE	Zosterops maderaspatanus	valid	Zosteropidae	White-eyes
Zosterops	302	valid/MALAGASY WHITE EYE/2.jpg	MALAGASY WHITE EYE	Zosterops maderaspatanus	valid	Zosteropidae	White-eyes
Zosterops	302	valid/MALAGASY WHITE EYE/3.jpg	MALAGASY WHITE EYE	Zosterops maderaspatanus	valid	Zosteropidae	White-eyes
Zosterops	302	valid/MALAGASY WHITE EYE/4.jpg	MALAGASY WHITE EYE	Zosterops maderaspatanus	valid	Zosteropidae	White-eyes
Zosterops	302	valid/MALAGASY WHITE EYE/5.jpg	MALAGASY WHITE EYE	Zosterops maderaspatanus	valid	Zosteropidae	White-eyes

Figure 14 Dataframe with the family names.

3. Data Product Code

The product code was split over three separate notebooks, one for training the model, one for creating the visualizations, and one for creating the user interface.

A. Model training notebook

In this notebook, we processed the data so that it was ready to be used to train the model. We also compared three convolutional neural networks (CNNs) and picked the one that gave the best results.

This notebook runs in a separate environment that has the GPU-enabled version of TensorFlow installed. This gives a significant speed boost when training the model. Because of compatibility issues, this unfortunately also means that matplotlib could not be installed in this environment.

Setting up the environment and reading the data

Setting up environment

```
# Import necessary tools
import tensorflow as tf
import tensorflow_hub as hub
import numpy as np
import pandas as pd
import os
import datetime
from IPython.display import Image

# Set up tensorboard
%load_ext tensorboard

# Check that GPU is available
if tf.config.list_physical_devices("GPU"):
    print("GPU available")
else:
    print("No GPU available")
```

```
GPU available
```

Figure 15 Importing the libraries.

Constant variables

```
# Image input size required by model
IMG_SIZE = 224

# Batch size
BATCH_SIZE = 32

# Model paths on tensorflowhub
RESNET = "https://tfhub.dev/google/imagenet/resnet_v2_50/classification/5"
MOBILENET = "https://tfhub.dev/google/imagenet/mobilenet_v3_large_075_224/classification/5"
EFFICIENTNET = "https://tfhub.dev/google/imagenet/efficientnet_v2_imagenet1k_b0/classification/2"

# Input shape for the model (batch, height, width, color channels)
INPUT_SHAPE = [None, IMG_SIZE, IMG_SIZE, 3]

# Output shape for the model (number of unique bird species)
OUTPUT_SHAPE = 450

# Epochs for training the model
EPOCHS = 100
```

Figure 16 Constants.

First, we read the dataset from the csv file.

Create labels and filepaths

Import data

```
birds = pd.read_csv("data/birds/birds.csv")
```

```
birds.info
```

```
<bound method DataFrame.info of        class id                          filepaths
labels  \
0             0        train/ABBOTTS BABBLER/001.jpg          ABBOTTS BABBLER
1             0        train/ABBOTTS BABBLER/002.jpg          ABBOTTS BABBLER
2             0        train/ABBOTTS BABBLER/003.jpg          ABBOTTS BABBLER
3             0        train/ABBOTTS BABBLER/004.jpg          ABBOTTS BABBLER
4             0        train/ABBOTTS BABBLER/005.jpg          ABBOTTS BABBLER
...         ...                                  ...                      ...
75121       449  valid/YELLOW HEADED BLACKBIRD/1.jpg  YELLOW HEADED BLACKBIRD
75122       449  valid/YELLOW HEADED BLACKBIRD/2.jpg  YELLOW HEADED BLACKBIRD
75123       449  valid/YELLOW HEADED BLACKBIRD/3.jpg  YELLOW HEADED BLACKBIRD
75124       449  valid/YELLOW HEADED BLACKBIRD/4.jpg  YELLOW HEADED BLACKBIRD
75125       449  valid/YELLOW HEADED BLACKBIRD/5.jpg  YELLOW HEADED BLACKBIRD

           scientific label data set
0      Malacocincla abbotti    train
1      Malacocincla abbotti    train
2      Malacocincla abbotti    train
3      Malacocincla abbotti    train
4      Malacocincla abbotti    train
...                     ...      ...
75121       Xanthocephalus    valid
75122       Xanthocephalus    valid
75123       Xanthocephalus    valid
75124       Xanthocephalus    valid
75125       Xanthocephalus    valid

[75126 rows x 5 columns]>
```

The birds dataset contains 75k images of birds, seperated into training, test and validation sets.

Figure 17 Reading in the bird csv file.

We make a list of unique labels (unique bird species).

Create unique labels array

This array holds all the different species that our model will be able to classify (450 unique bird species).

```
In [15]: unique_labels = np.asarray(birds["labels"].unique())
         unique_labels
```

```
Out[15]: array(['ABBOTTS BABBLER', 'ABBOTTS BOOBY', 'ABYSSINIAN GROUND HORNBILL',
                'AFRICAN CROWNED CRANE', 'AFRICAN EMERALD CUCKOO',
                'AFRICAN FIREFINCH', 'AFRICAN OYSTER CATCHER',
                'AFRICAN PIED HORNBILL', 'ALBATROSS', 'ALBERTS TOWHEE',
                'ALEXANDRINE PARAKEET', 'ALPINE CHOUGH', 'ALTAMIRA YELLOWTHROAT',
                'AMERICAN AVOCET', 'AMERICAN BITTERN', 'AMERICAN COOT',
                'AMERICAN FLAMINGO', 'AMERICAN GOLDFINCH', 'AMERICAN KESTREL',
                'AMERICAN PIPIT', 'AMERICAN REDSTART', 'AMERICAN WIGEON',
                'AMETHYST WOODSTAR', 'ANDEAN GOOSE', 'ANDEAN LAPWING',
                'ANDEAN SISKIN', 'ANHINGA', 'ANIANIAU', 'ANNAS HUMMINGBIRD',
                'ANTBIRD', 'ANTILLEAN EUPHONIA', 'APAPANE', 'APOSTLEBIRD',
                'ARARIPE MANAKIN', 'ASHY STORM PETREL', 'ASHY THRUSHBIRD',
                'ASIAN CRESTED IBIS', 'ASIAN DOLLARD BIRD', 'AUCKLAND SHAQ',
                'AUSTRAL CANASTERO', 'AUSTRALASIAN FIGBIRD', 'AVADAVAT',
                'AZARAS SPINETAIL', 'AZURE BREASTED PITTA', 'AZURE JAY',
                'AZURE TANAGER', 'AZURE TIT', 'BAIKAL TEAL', 'BALD EAGLE',
                'BALD IBIS', 'BALI STARLING', 'BALTIMORE ORIOLE', 'BANANAQUIT',
                'BAND TAILED GUAN', 'BANDED BROADBILL', 'BANDED PITA',
                'BANDED STILT', 'BAR-TAILED GODWIT', 'BARN OWL', 'BARN SWALLOW',
```

```
In [16]: len(unique_labels)
```

```
Out[16]: 450
```

Figure 18 Create unique labels list.

From the birds dataframe, we can extract the filepaths to the images and the corresponding labels. We convert the labels to Boolean form.

Create filepaths and labels

```
filepaths = ["data/birds/" + path for path in birds["filepaths"]]
filepaths = np.array(filepaths)
filepaths[:5]
```

```
array(['data/birds/train/ABBOTTS BABBLER/001.jpg',
       'data/birds/train/ABBOTTS BABBLER/002.jpg',
       'data/birds/train/ABBOTTS BABBLER/003.jpg',
       'data/birds/train/ABBOTTS BABBLER/004.jpg',
       'data/birds/train/ABBOTTS BABBLER/005.jpg'], dtype='<U54')
```

```
labels = [label for label in birds["labels"]]
labells = np.array(labels)
labels[:5]
```

```
['ABBOTTS BABBLER',
 'ABBOTTS BABBLER',
 'ABBOTTS BABBLER',
 'ABBOTTS BABBLER',
 'ABBOTTS BABBLER']
```

```
# Turn labels into boolean labels
boolean_labels = [label == unique_labels for label in labels]
boolean_labels = np.array(boolean_labels)
boolean_labels[0][:10]
```

```
array([ True, False, False, False, False, False, False, False, False,
       False])
```

Figure 19 Creation of filepaths, labels, and boolean labels.

We split the dataset into a training set, a test set, and a validation set. For development purposes, we also create a smaller subset of the training set (1000 images) which we used to try out training the model, without having to wait too long for the results.

Creating a training, test and validation set

Because our full data set is large, we will use a smaller subset to test our model on first before training on all the data. Since the test and validation sets are 2k each, we can use them as they are.

Training set

First we create the full taining set.

```
# Extract the training set
full_training_set = birds.loc[birds['data set'] == "train"]
full_training_set.to_numpy()
full_training_set.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 70626 entries, 0 to 70625
Data columns (total 5 columns):
 #   Column            Non-Null Count  Dtype 
---  ------            --------------  ----- 
 0   class id          70626 non-null  int64 
 1   filepaths         70626 non-null  object
 2   labels            70626 non-null  object
 3   scientific label  70626 non-null  object
 4   data set          70626 non-null  object
dtypes: int64(1), object(4)
memory usage: 3.2+ MB
```

```
# scramble full training dataset
np.random.seed(7)
scrambled_full_train = full_training_set.sample(frac=1, ignore_index=True)
scrambled_full_train.to_numpy();
```

```
# Create full filepath set
full_train_filepaths = ["data/birds/" + path for path in scrambled_full_train["filepaths"]]

# Create full labels set
full_train_labels = [label for label in scrambled_full_train["labels"]]

# Turn labels set in boolean labels
full_train_bool_labels = [label == unique_labels for label in full_train_labels]
```

```
# Check the lengths of the filepaths, labels and boolean labels
print(len(full_train_filepaths), len(full_train_labels), len(full_train_bool_labels))
```

```
70626 70626 70626
```

Now we create a smaller (1000 images) subset.

```
# Create smaller subset
small_train_filepaths = full_train_filepaths[:1000]
small_train_labels = full_train_labels[:1000]
small_train_bool_labels = full_train_bool_labels[:1000]
```

```
# Check the lengths
print(len(small_train_filepaths), len(small_train_labels), len(small_train_bool_labels))
```

```
1000 1000 1000
```

Figure 20 Creation of full training subset and smaller training subset.

We do the same for the test set and the validation set.

Test set

```
# Extract the test set
test_set = birds.loc[birds['data set'] == "test"]
test_set.to_numpy()
test_set.info
```

```
<bound method DataFrame.info of        class id                          filepaths
70626         0          test/ABBOTTS BABBLER/1.jpg          ABBOTTS BABBLER
70627         0          test/ABBOTTS BABBLER/2.jpg          ABBOTTS BABBLER
70628         0          test/ABBOTTS BABBLER/3.jpg          ABBOTTS BABBLER
70629         0          test/ABBOTTS BABBLER/4.jpg          ABBOTTS BABBLER
70630         0          test/ABBOTTS BABBLER/5.jpg          ABBOTTS BABBLER
...         ...                                 ...                        ...
72871       449  test/YELLOW HEADED BLACKBIRD/1.jpg  YELLOW HEADED BLACKBIRD
72872       449  test/YELLOW HEADED BLACKBIRD/2.jpg  YELLOW HEADED BLACKBIRD
72873       449  test/YELLOW HEADED BLACKBIRD/3.jpg  YELLOW HEADED BLACKBIRD
72874       449  test/YELLOW HEADED BLACKBIRD/4.jpg  YELLOW HEADED BLACKBIRD
72875       449  test/YELLOW HEADED BLACKBIRD/5.jpg  YELLOW HEADED BLACKBIRD

          scientific label data set
70626  Malacocincla abbotti     test
70627  Malacocincla abbotti     test
70628  Malacocincla abbotti     test
70629  Malacocincla abbotti     test
70630  Malacocincla abbotti     test
```

```
# scramble test set
np.random.seed(7)
scrambled_test = test_set.sample(frac=1,
                                 ignore_index=True)
scrambled_test.to_numpy();
```

```
# Create a test filepath array
test_filepaths = ["data/birds/" + path for path in scrambled_test["filepaths"]]

# Create a test labels array
test_labels = [label for label in scrambled_test["labels"]]

# Create boolean test labels array
test_bool_labels = [label == unique_labels for label in test_labels]
```

```
# Check the lengths
print(len(test_filepaths), len(test_labels), len(test_bool_labels))
```

```
2250 2250 2250
```

Figure 21 Creation of test set.

Validation set

```
# Extract the validation set
val_set = birds.loc[birds['data set'] == "valid"]
val_set.to_numpy()
val_set.info
```

```
<bound method DataFrame.info of        class id                          filepaths
72876         0          valid/ABBOTTS BABBLER/1.jpg          ABBOTTS BABBLER
72877         0          valid/ABBOTTS BABBLER/2.jpg          ABBOTTS BABBLER
72878         0          valid/ABBOTTS BABBLER/3.jpg          ABBOTTS BABBLER
72879         0          valid/ABBOTTS BABBLER/4.jpg          ABBOTTS BABBLER
72880         0          valid/ABBOTTS BABBLER/5.jpg          ABBOTTS BABBLER
...         ...                                 ...                        ...
75121       449  valid/YELLOW HEADED BLACKBIRD/1.jpg  YELLOW HEADED BLACKBIRD
75122       449  valid/YELLOW HEADED BLACKBIRD/2.jpg  YELLOW HEADED BLACKBIRD
75123       449  valid/YELLOW HEADED BLACKBIRD/3.jpg  YELLOW HEADED BLACKBIRD
75124       449  valid/YELLOW HEADED BLACKBIRD/4.jpg  YELLOW HEADED BLACKBIRD
75125       449  valid/YELLOW HEADED BLACKBIRD/5.jpg  YELLOW HEADED BLACKBIRD

          scientific label data set
72876  Malacocincla abbotti     valid
72877  Malacocincla abbotti     valid
72878  Malacocincla abbotti     valid
72879  Malacocincla abbotti     valid
72880  Malacocincla abbotti     valid
```

```
# scramble validation set
np.random.seed(7)
scrambled_val = val_set.sample(frac=1,
                               ignore_index=True)
scrambled_val.to_numpy();
```

```
# Create a validation filepath array
val_filepaths = ["data/birds/" + path for path in scrambled_val["filepaths"]]
# Create a validation labels array
val_labels = [label for label in scrambled_val["labels"]]
# Create boolean validation labels array
val_bool_labels = [label == unique_labels for label in val_labels]
```

```
# Check the lengths
print(len(val_filepaths), len(val_labels), len(val_bool_labels))
```

```
2250 2250 2250
```

Figure 22 Creation of validation set.

Preprocessing the data

TensorFlow is optimized to work best when the data is provided to it in batches. We will use the function batchify() to convert the images to tensors and then group them in batches of 32. The batchify function in turn uses the function image_to_tensor(), which reads the image, decodes it, converts it to numeric form, and resizes it to the correct size (224px by 224 px).

Preprocess the input for the model

```
# Create training batches (small training set)
small_train_batch = batchify(filepaths=small_train_filepaths, labels=small_train_bool_labels)
small_train_batch.element_spec
# Tuple of image (224, 224, 3) and lable (boolean of 450 species)
```

```
(TensorSpec(shape=(None, 224, 224, 3), dtype=tf.float32, name=None),
 TensorSpec(shape=(None, 450), dtype=tf.bool, name=None))
```

```
# Create training batches (full training set)
full_train_batch = batchify(filepaths=full_train_filepaths, labels=full_train_bool_labels)
# Create testing batches
test_batch = batchify(filepaths=test_filepaths, labels=test_bool_labels)
# Create validation batches
val_batch = batchify(filepaths=val_filepaths, labels=val_bool_labels)
```

We have now set up the following data batches ready to use in our model

- small_train_batch
- full_train_batch
- test_batch
- val_batch

Figure 23 Preprocessing the images.

```python
def image_to_tensor(path, label=None, img_size=IMG_SIZE):
    """
    Takes an image, transforms it to a tensor and resizes it if needed.
    If a label is provided, it returns a tuple (image, label), otherwise it returns the transformed image alone.
    """
    img = tf.io.read_file(path)
    img = tf.image.decode_jpeg(img, channels=3)
    img = tf.image.convert_image_dtype(img, tf.float32)
    img = tf.image.resize(img, size=[img_size, img_size])

    if label is not None:
        return (img, label)
    else:
        return img
```

```python
def batchify(filepaths, labels=None, batch_size=BATCH_SIZE):
    """
    This function turns a set of data (filepath and label) into batches.
    """
    if labels is None:    # Only images
        data = tf.data.Dataset.from_tensor_slices((tf.constant(filepaths))) # Only images
    else:                 # Images and labels
        data = tf.data.Dataset.from_tensor_slices((tf.constant(filepaths),
                                                   tf.constant(labels)))
    data = data.map(image_to_tensor)    # Transform to tensors
    data_batch = data.batch(batch_size) # Turn into batches
    return data_batch
```

Figure 24 Functions image_to_tensor() and batchify().

Training the models

Now we are ready to train the model. We will train and compare three pre-trained models. All three of these models have been pre-trained on the ImageNet (ILSVRC-2012-CLS) dataset (Russakovsky et al., 2015) to classify general images.

- ResNet50: a 50-layer convolutional neural network. (He et al., 2015) https://tfhub.dev/google/imagenet/resnet_v2_50/classification/5
- MobileNetV3: a lightweight CNN. (Howard et al., 2019) https://tfhub.dev/google/imagenet/mobilenet_v3_large_075_224/classification/5
- EfficientNet: a CNN optimized for faster training. (Tan & Le, 2021) https://tfhub.dev/google/imagenet/efficientnet_v2_imagenet1k_b0/classification/2

The create_and_train_model() function calls the create_model() function to create the model. Then it creates the callbacks by calling the create_tensorboard_callback() and the create_early_stopping_callback() functions. Lastly, it fits the model on the training data and uses the validation data to evaluate the model. Then it returns the whole model.

```
def create_and_train_model(train_data=None, val_data=None,
                           input_shape=INPUT_SHAPE, output_shape=OUTPUT_SHAPE, model=MOBILENET,
                           log_path="E:/Kathleen/Capstone/logs/", patience=3, epochs=EPOCHS):
    """
    Creates and trains a model. It takes the training and testing data in batch form, the input and output shape
    of the model, the model url, the path where logs will be stored and the patience for the early stopping callback.
    """
    # Create model
    model = create_model(input_shape=input_shape, output_shape=output_shape, model=model)

    # Create tensorboard and early stopping callbacks
    tensorboard = create_tensorboard_callback(log_path)
    early_stopping = create_early_stopping_callback(patience)


    # Fit the model
    model = fit_model(model,
                      x=train_data,
                      val_data=val_data,
                      callbacks=[tensorboard, early_stopping])

    return model
```

Figure 25 The create_and_train_model() function.

The create_model() function creates the model. It defines the input and output layers. It also compiles the model. For the loss function, we use CategoricalCrossentropy. We use the Adam optimizer. For metrics, we track the accuracy, the precision, and the recall. Then we build the model and define the input shape.

```
def create_model(input_shape=INPUT_SHAPE, output_shape=OUTPUT_SHAPE, model=MOBILENET):
    """
    This function builds the model based on the given input and output shapes and the url to the
    prelearned model. The model is build so that it tracks accuracy, precision and recall.
    """
    model = tf.keras.Sequential([hub.KerasLayer(model),    # input layer = pretrained model
                                 tf.keras.layers.Dense(units=OUTPUT_SHAPE, activation="softmax")]) # Output layer
    model.compile(loss=tf.keras.losses.CategoricalCrossentropy(),
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=["accuracy",   # Track 3 metrics
                           tf.keras.metrics.Precision(),
                           tf.keras.metrics.Recall()])
    model.build(INPUT_SHAPE)
    return model
```

Figure 26 The create_model() function.

The create_tensorboard_callback() function creates the callback so that we can view the metrics of the training process on the tensorboard afterward.

The create_early_stopping_callback() function creates the callback that stops the training process when the validation accuracy no longer improves. This can help prevent overfitting the model. It has a standard patience of 3, meaning that 3 consecutive val_accuracies need to be lower than the one before for the callback to activate.

```
def create_tensorboard_callback(path="E:/Kathleen/Capstone/logs/"):
    """
    Creates a tensorboard callback.
    """
    logdir = os.path.join(path, datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))  # Adds a timestamp to logs
    callback = tf.keras.callbacks.TensorBoard(log_dir=logdir, histogram_freq=1)
    return callback
```

```
def create_early_stopping_callback(patience=3):
    """
    Creates a callback for early stopping if the val_accuracy no longer improves during training.
    """
    callback = tf.keras.callbacks.EarlyStopping(monitor="val_accuracy", patience=patience)
    return callback
```

Figure 27 The create_tensorboard_callback() and create_early_stopping_callback() functions.

The fit_model() function takes the model, the training data (x), the validation data, the callbacks, and the number of epochs and fits the model. Then it returns the fitted model.

```
def fit_model(model, x, val_data, callbacks=[], epochs=EPOCHS, val_freq=1):
    """
    Fits the model with the given data and callbacks.
    """
    model.fit(x=x,
              epochs=epochs,
              validation_data=val_data,
              validation_freq=val_freq,
              callbacks=callbacks)
    return model
```

Figure 28 The fit_model() function.

The save_model() and load_model() functions save and load the trained model so we can reuse them later or in other notebooks.

```
def save_model(model, path="./models/", suffix=""):
    """
    Saves a model.
    """
    modeldir = os.path.join(path,datetime.datetime.now().strftime("%Y%m%d-%H%M"))
    model_path = modeldir + "-" + suffix + ".h5"
    model.save(model_path)
    return model_path
```

```
def load_model(path):
    """
    Loads a saved model.
    """
    model = tf.keras.models.load_model(path, custom_objects={"KerasLayer":hub.KerasLayer})
    return model
```

Figure 29 The save_model() and load_model() functions.

First, we train a ResNet model. We can see here that we get a final validation accuracy of 89%, just below our target of a minimum of 90% accuracy. The training lasted for 17 epochs before the early_stopping_callback decided the accuracy was no longer improving. Each epoch took around 131s, for a total training time of 37 minutes. The trained ResNet model is 103MB. We will evaluate and compare the models in more detail later.

ResNet50

```
In [39]: full_model_resnet = create_and_train_model(model=RESNET, train_data=full_train_batch, val_data=val_batch)
0.9646 - val_loss: 0.6072 - val_accuracy: 0.8964 - val_precision: 0.9039 - val_recall: 0.8907
Epoch 12/100
2208/2208 [==============================] - 130s 59ms/step - loss: 0.1032 - accuracy: 0.9678 - precision: 0.9718 - recall:
0.9646 - val_loss: 0.6559 - val_accuracy: 0.8889 - val_precision: 0.8980 - val_recall: 0.8840
Epoch 13/100
2208/2208 [==============================] - 131s 59ms/step - loss: 0.0916 - accuracy: 0.9708 - precision: 0.9744 - recall:
0.9680 - val_loss: 0.6563 - val_accuracy: 0.8929 - val_precision: 0.8997 - val_recall: 0.8893
Epoch 14/100
2208/2208 [==============================] - 131s 59ms/step - loss: 0.0823 - accuracy: 0.9734 - precision: 0.9764 - recall:
0.9710 - val_loss: 0.6453 - val_accuracy: 0.9031 - val_precision: 0.9094 - val_recall: 0.9013
Epoch 15/100
2208/2208 [==============================] - 130s 59ms/step - loss: 0.0765 - accuracy: 0.9757 - precision: 0.9783 - recall:
0.9735 - val_loss: 0.6900 - val_accuracy: 0.8916 - val_precision: 0.8993 - val_recall: 0.8889
Epoch 16/100
2208/2208 [==============================] - 134s 61ms/step - loss: 0.0720 - accuracy: 0.9775 - precision: 0.9795 - recall:
0.9753 - val_loss: 0.7051 - val_accuracy: 0.8947 - val_precision: 0.9018 - val_recall: 0.8893
Epoch 17/100
2208/2208 [==============================] - 134s 60ms/step - loss: 0.0664 - accuracy: 0.9790 - precision: 0.9810 - recall:
0.9776 - val_loss: 0.7196 - val_accuracy: 0.8920 - val_precision: 0.8992 - val_recall: 0.8880

In [40]: save_model(full_model_resnet, suffix="ResNet")
Out[40]: './models/20221103-1834-ResNet.h5'
```

Figure 30 Training the ResNet model.

The next model is the MobileNet model. We get a final val_accuracy of 93%, which is above our goal of 90%. The training lasted for only 9 epochs, and each epoch took roughly 75s, for a total of 11 minutes. The trained model is 20MB.

MobileNet V3

```
full_model_mobilenet = create_and_train_model(model=MOBILENET, train_data=full_train_batch, val_data=val_batch)
Epoch 1/100
2208/2208 [==============================] - 84s 35ms/step - loss: 0.8417 - accuracy: 0.8190 - precision_1: 0.9312 - recall_1:
0.7298 - val_loss: 0.2546 - val_accuracy: 0.9271 - val_precision_1: 0.9511 - val_recall_1: 0.9071
Epoch 2/100
2208/2208 [==============================] - 76s 35ms/step - loss: 0.2374 - accuracy: 0.9379 - precision_1: 0.9609 - recall_1:
0.9213 - val_loss: 0.2330 - val_accuracy: 0.9333 - val_precision_1: 0.9509 - val_recall_1: 0.9218
Epoch 3/100
2208/2208 [==============================] - 76s 34ms/step - loss: 0.1288 - accuracy: 0.9677 - precision_1: 0.9786 - recall_1:
0.9586 - val_loss: 0.2449 - val_accuracy: 0.9342 - val_precision_1: 0.9496 - val_recall_1: 0.9298
Epoch 4/100
2208/2208 [==============================] - 74s 33ms/step - loss: 0.0786 - accuracy: 0.9813 - precision_1: 0.9863 - recall_1:
0.9764 - val_loss: 0.2464 - val_accuracy: 0.9351 - val_precision_1: 0.9440 - val_recall_1: 0.9284
Epoch 5/100
2208/2208 [==============================] - 75s 34ms/step - loss: 0.0594 - accuracy: 0.9848 - precision_1: 0.9878 - recall_1:
0.9811 - val_loss: 0.2515 - val_accuracy: 0.9360 - val_precision_1: 0.9433 - val_recall_1: 0.9316
Epoch 6/100
2208/2208 [==============================] - 75s 34ms/step - loss: 0.0456 - accuracy: 0.9880 - precision_1: 0.9898 - recall_1:
0.9856 - val_loss: 0.2451 - val_accuracy: 0.9387 - val_precision_1: 0.9477 - val_recall_1: 0.9347
Epoch 7/100
2208/2208 [==============================] - 73s 33ms/step - loss: 0.0369 - accuracy: 0.9897 - precision_1: 0.9910 - recall_1:
0.9883 - val_loss: 0.2705 - val_accuracy: 0.9342 - val_precision_1: 0.9405 - val_recall_1: 0.9280
Epoch 8/100
2208/2208 [==============================] - 75s 34ms/step - loss: 0.0283 - accuracy: 0.9922 - precision_1: 0.9930 - recall_1:
0.9912 - val_loss: 0.2934 - val_accuracy: 0.9387 - val_precision_1: 0.9453 - val_recall_1: 0.9364
Epoch 9/100
2208/2208 [==============================] - 75s 34ms/step - loss: 0.0255 - accuracy: 0.9926 - precision_1: 0.9932 - recall_1:
0.9917 - val_loss: 0.2871 - val_accuracy: 0.9347 - val_precision_1: 0.9404 - val_recall_1: 0.9324

save_model(full_model_mobilenet, suffix="MobileNet")
'./models/20221103-1845-MobileNet.h5'
```

Figure 31 Training the MobileNet model.

The last model is EfficientNet. We get a final val_accuracy of 91%, which is above our goal of 90%. The training lasted for 21 epochs, and each epoch took roughly 98s, for a total of 34 minutes (even though this model is supposed to be optimized for faster training). The trained model is 33MB.

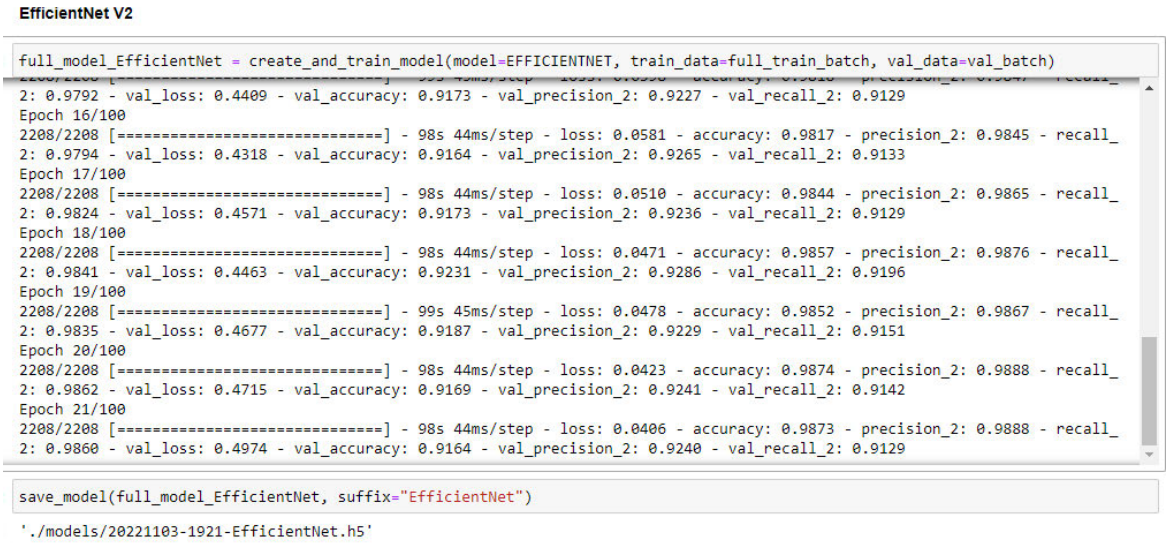


Figure 32 Training the EfficientNet model.

Comparing the models

After training is complete, we can check the Tensorboard to see the metrics of the three models. We are only interested in the metrics of the validation data, not the training data.

Accuracy:

MobileNet achieves the highest final accuracy with 93%. Next is EfficientNet with 91% and lastly ResNet with an accuracy of 89%.

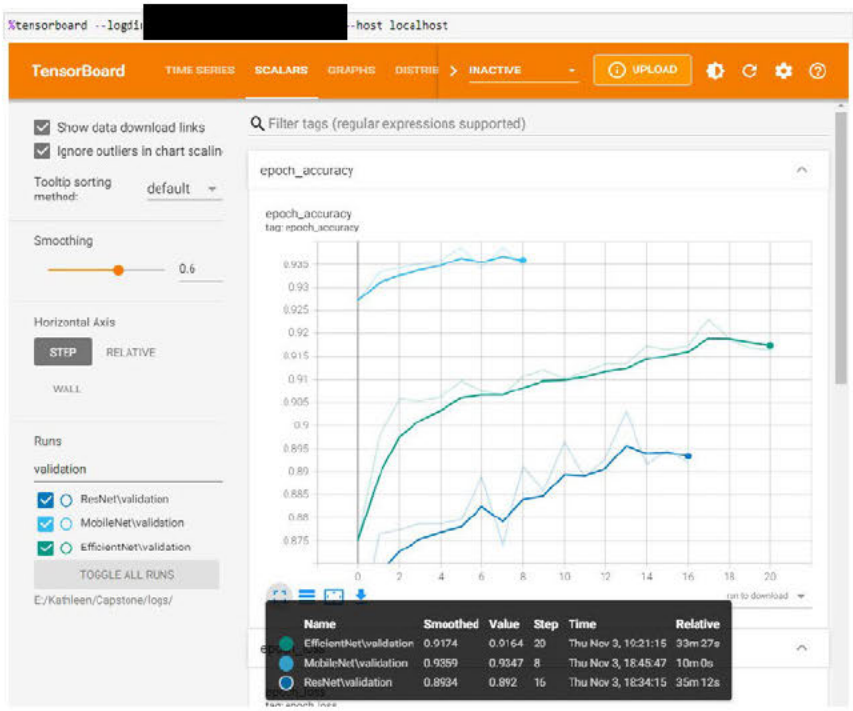


Figure 33 Accuracy for the three models.

Loss:

MobileNet does best for loss as well, with a loss of 0.27. Next comes EfficientNet with a loss of 0.47 and lastly ResNet with a loss of 0.69.

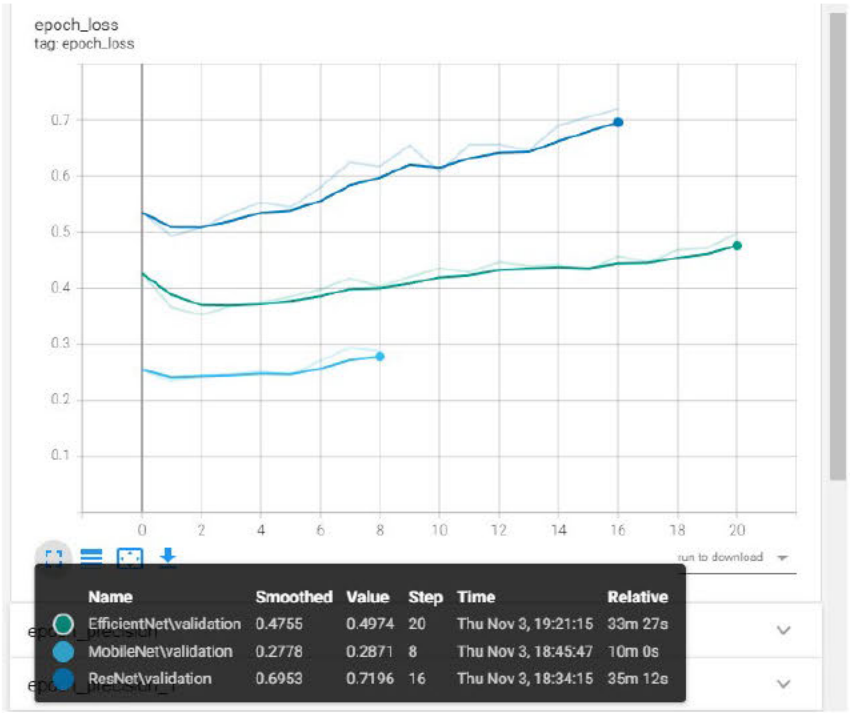


Figure 34 Loss for the three models.

Precision:

MobileNet has a final precision of 94%, ResNet reaches 90%, and EfficientNet reaches 92%.

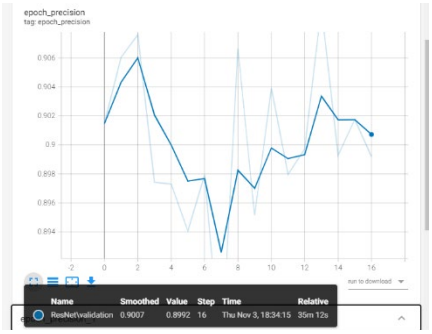


Figure 35 Precision of ResNet.

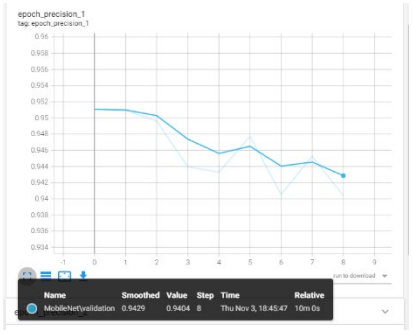


Figure 36 Precision of MobileNet

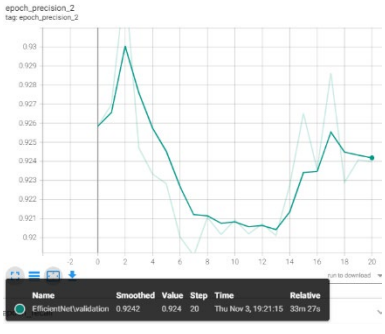


Figure 37 Precision of EfficientNet

Recall:

MobileNet has a final recall of 93%, ResNet reaches 88%, and EfficientNet has 91%.

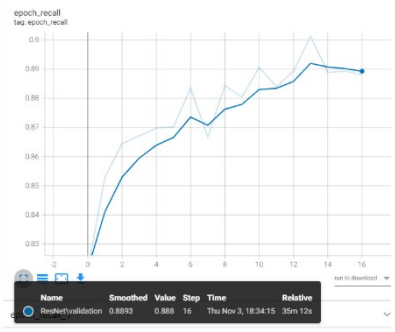


Figure 35 Recall of ResNet.

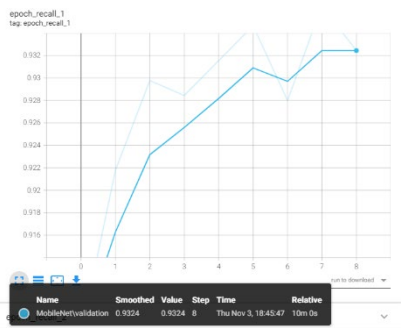


Figure 36 Recall of MobileNet.

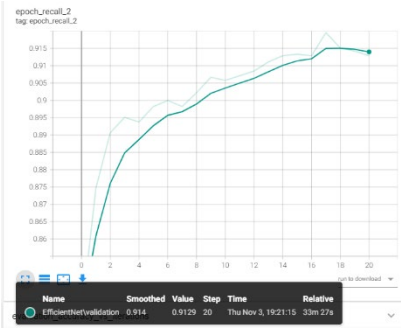


Figure 37 Recall of EfficientNet.

The above graphs compare the three models based on the validation dataset. We can also compare them using the testing dataset.

Comparing the models

```
# Load previously trained models
model_resnet = load_model('E:/Kathleen/Capstone/models/ResNet.h5')
model_mobilenet = load_model('E:/Kathleen/Capstone/models/MobileNet.h5')
model_efficientnet = load_model('E:/Kathleen/Capstone/models/EfficientNet.h5')
```

```
WARNING:tensorflow:Please fix your imports. Module tensorflow.python.training.tracking.data_structures has been moved to tensor
flow.python.trackable.data_structures. The old module will be deleted in version 2.11.
```

```
# Get metrics for each model
resnet_scores = model_resnet.evaluate(test_batch)
mobilenet_scores = model_mobilenet.evaluate(test_batch)
efficientnet_scores = model_efficientnet.evaluate(test_batch)
```

```
71/71 [==============================] - 7s 57ms/step - loss: 0.5096 - accuracy: 0.9098 - precision: 0.9154 - recall: 0.9089
71/71 [==============================] - 3s 33ms/step - loss: 0.2085 - accuracy: 0.9542 - precision_1: 0.9579 - recall_1: 0.951
6
71/71 [==============================] - 4s 41ms/step - loss: 0.4075 - accuracy: 0.9280 - precision_2: 0.9325 - recall_2: 0.927
1
```

```
# Display the results in a dataframe for easy comparison
comparison = pd.DataFrame(
    {'ResNet': resnet_scores,
     'MobileNet': mobilenet_scores,
     'EfficientNet': efficientnet_scores
    })
comparison.index = ['Loss', 'Accuracy', 'Precision', 'Recall']
comparison.T
```

	Loss	Accuracy	Precision	Recall
ResNet	0.509602	0.909778	0.915398	0.908889
MobileNet	0.208484	0.954222	0.957942	0.951556
EfficientNet	0.407527	0.928000	0.932499	0.927111

From the comparison dataframe, we can see that the mobilenet model outperforms the other two models in all metrics. The loss is lower, the accuracy is higher, the precision is higher and the recall is higher. When we compare the size of the saved models, we see that ResNet takes up 103MB, EfficientNet 32.9MB and MobileNet 20.7MB. We therefore pick MobileNet for our application.

Figure 38 Comparison of the three models.

We see that the MobileNet does best in all metrics, and takes up the least amount of memory, so the decision of which model to use for our application is easy.

B. Visualization notebook

This notebook is used to visualize the dataset and the predictions made by the model.

Setting up the environment and loading up the data.

First, we set up our imports and constants.

Visualizations

```
This notebook visualizes the data we have as input or output of our model.

Because of compatibility problems, we cannot use a GPU when we have matlibplot installed, so the training of the model will be
done in a seperate notebook and in this notebook, we will import the trained model.
```

```
# Import necessary tools
import tensorflow as tf
import tensorflow_hub as hub
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import os
import datetime
import ipywidgets as widgets
import PIL.Image
from IPython.display import display, clear_output
from IPython.display import Image
from statistics import mean

%matplotlib inline

# !jupyter nbextension enable --py widgetsnbextension --sys-prefix
# !jupyter serverextension enable voila --sys-prefix

# Warning work-arounds
os.environ["KMP_DUPLICATE_LIB_OK"]="True"
pd.options.mode.chained_assignment = None
```

Constants

```
# Image input size required by model
IMG_SIZE = 224

# Batch size
BATCH_SIZE = 32
```

Figure 39 Imports and constants.

We read the data into the notebook and convert it to filepaths and labels in the same way as we did in the model training notebook. We can visually explore some of the images of the dataset to get an idea of what we are working with.

For this, we can use three functions. Show_image() reads a filepath and displays the image and the label (if given). Show_num_image() takes an image that is in numeric format and displays it with the label (if given). Show_multiple_images() takes a list, the number of columns you want to show them in, and the size the images should be and displays a grid of images with the labels above them. If the label is missing, it shows “Unknown label” above the image.

```python
def show_image(pathname, label=None):
    """
    This function displays an image when given its path. If a label is given, it will appear above the image.
    """
    img = mpimg.imread(pathname)
    plt.xticks([])
    plt.yticks([])
    plt.title(label)
    plt.imshow(img)
```

```python
def show_num_image(image, label=None):
    """
    This function displays an image when given in numeric format. If a label is given, it will appear above the image.
    """
    plt.xticks([])
    plt.yticks([])
    plt.title(label)
    plt.imshow(image)
```

```python
def show_multiple_images(list, columns=3, size=10):
    """
    This function takes a list of tuples (file path, label) representing images and displays them in a grid of
    3 images wide, with the label on top (if given).
    """
    images = []
    labels = []
    for tuple in list:
        images.append(mpimg.imread(tuple[0])) # transform to numerics
        try:
            labels.append(tuple[1])           # Label was given
        except IndexError:
            labels.append('Unknown label')    # No label was given

    plt.figure(figsize=(size, size))
    for i in range(len(images)):              # Add each image to the roster
        ax = plt.subplot(np.ceil(len(images)/columns).astype(int), columns, i+1)
        plt.imshow(images[i])
        plt.title(labels[i])
        plt.xticks([])
        plt.yticks([])
        plt.axis("off")
    plt.tight_layout(h_pad=1.0)
    plt.show()
```

Figure 40 Functions show_image(), show_num_image() and show_multiple_images().

Visualize some of the images

```python
#Check some of the images
show_image(filepaths[1000], labels[1000])
```

Figure 41 Visualization of a single image.

```python
images = [(filepaths[10000],labels[10000]),
          (filepaths[20000],labels[20000]),
          (filepaths[30000],labels[30000]),
          (filepaths[40000],),
          (filepaths[50000],labels[50000]),
          (filepaths[60000],labels[60000]),
          (filepaths[70000],labels[70000])]
show_multiple_images(images, columns =3, size=6)
```

Figure 42 Visualization of multiple images.

Next, we check the number of images in the different subsets (training, testing, and validation). We see that the validation and test sets both have exactly 5 images per species. The training set has between 248 and 130 images per species, so the total set has between 258 and 140 images per species. Each species has at least 100 images, so we should get a decent accuracy in our model.

Check number of images per species

```
# Number of images in training set per species
birds.loc[birds['data set'] == "train"]["labels"].value_counts()
```

```
HOUSE FINCH                248
D-ARNAUDS BARBET           233
OVENBIRD                   233
SWINHOES PHEASANT          217
WOOD DUCK                  214
                          ...
GO AWAY BIRD               131
BLACK FRANCOLIN            131
RED TAILED THRUSH          130
PATAGONIAN SIERRA FINCH    130
SNOWY PLOVER               130
Name: labels, Length: 450, dtype: int64
```

Figure 43 Number of images per species in the training set.

```
# Number of images in test set per species
birds.loc[birds['data set'] == "test"]["labels"].value_counts()
```

```
ABBOTTS BABBLER            5
PARADISE TANAGER           5
MARABOU STORK              5
MANGROVE CUCKOO            5
MANDRIN DUCK               5
                          ..
CRANE HAWK                 5
CRAB PLOVER                5
COPPERY TAILED COUCAL      5
COMMON STARLING            5
YELLOW HEADED BLACKBIRD    5
Name: labels, Length: 450, dtype: int64
```

Figure 44 Number of images per species in the test set.

```
# Number of images in validation set per species
birds.loc[birds['data set'] == "valid"]["labels"].value_counts()
```

```
ABBOTTS BABBLER            5
PARADISE TANAGER           5
MARABOU STORK              5
MANGROVE CUCKOO            5
MANDRIN DUCK               5
                          ..
CRANE HAWK                 5
CRAB PLOVER                5
COPPERY TAILED COUCAL      5
COMMON STARLING            5
YELLOW HEADED BLACKBIRD    5
Name: labels, Length: 450, dtype: int64
```

Figure 45 Number of images in the validation set.

```
# Total (all data sets)
birds["labels"].value_counts()
```

```
HOUSE FINCH                258
D-ARNAUDS BARBET           243
OVENBIRD                   243
SWINHOES PHEASANT          227
WOOD DUCK                  224
                          ...
GO AWAY BIRD               141
BLACK FRANCOLIN            141
RED TAILED THRUSH          140
PATAGONIAN SIERRA FINCH    140
SNOWY PLOVER               140
Name: labels, Length: 450, dtype: int64
```

Figure 46 Number of images in the whole dataset.

Visualization 1: Bar plot of images per species.

Our first visual is a bar plot that shows the number of images per species.

1. Barplot of images per species

```
# Prepare data
labels = np.asarray(unique_labels)
values = np.asarray(birds["labels"].value_counts(sort=False))
species_count = pd.DataFrame(zip(labels, values), columns = ["species", "count"])
species_count = species_count.sort_values(by=['count'], axis=0, ascending=False)

# Create bar plot
fig, ax = plt.subplots(figsize = (50, 20))
species_bar = ax.bar(x=species_count['species'], height=species_count['count'])
fig.suptitle('Number of images in the dataset per bird species', fontsize=50)
plt.xticks(rotation='90', fontsize=10)
ax.set_xlim(0,450)

# Annotate max and min image count
max = species_count['count'].max()
min = species_count['count'].min()
ax.axhline(y=max, color='r', linestyle='--', label=max, linewidth=4)
ax.annotate(text="Max: " + str(max), xy=(0, max), xytext=(0, max), color='r', fontsize=50)
ax.axhline(y=min, color='black', linestyle='--', label=min, linewidth=4)
ax.annotate(text="Min: " + str(min), xy=(0, min), xytext=(450, min), color='black', fontsize=50)

# Display and save plot
plt.tight_layout()
plt.savefig('species_bar_plot.jpeg')
plt.show()
```

Figure 47 Code to create a barplot of images per species.

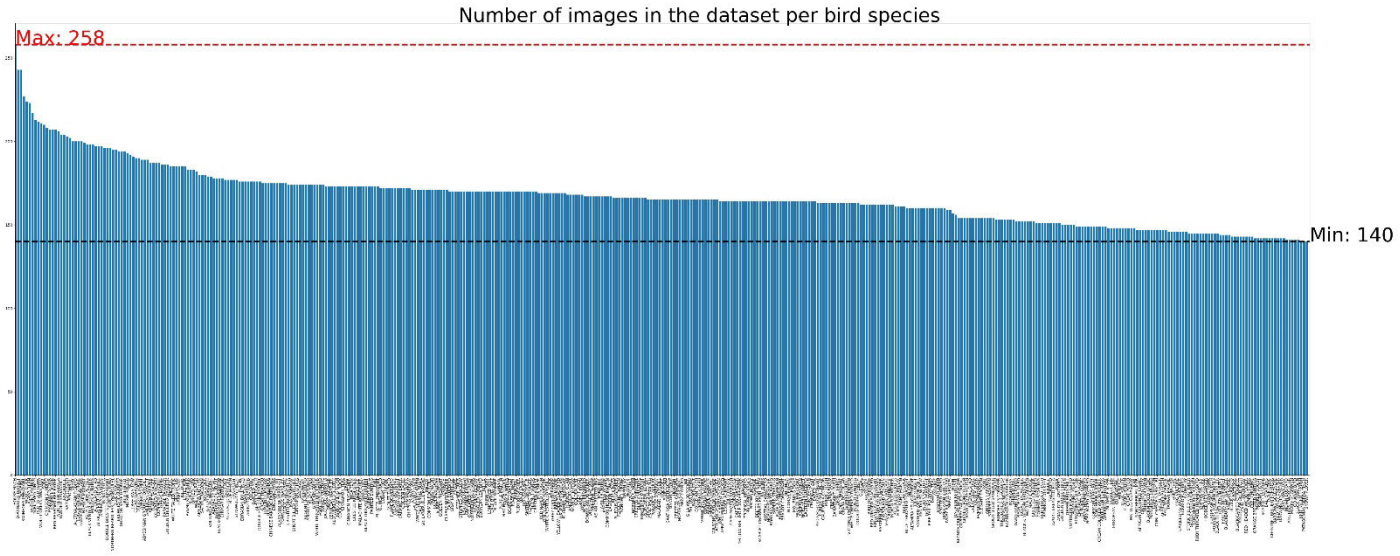


Figure 48 Bar plot of images per species.

Visualization 2: Scatter plot of images per family.

The second visual is a scatter plot representing the number of images per family. We have discussed how we added the family column to the birds dataframe in section D.2. Now we will use this dataframe to create a scatterplot that displays the number of images per family of birds.

First, we count the number of images per family.

Count number of images per family

```
# Count the number of images per family
labels = np.asarray(joined_birds["Family Name"].unique())
values = np.asarray(joined_birds["Family Name"].value_counts(sort=False))
family_count = pd.DataFrame(zip(labels, values), columns = ["family name", "count"])
family_count = family_count.sort_values(by=['count'], axis=0, ascending=False)
family_count
```

	family name	count
11	Pheasants	3583
10	Ducks	3361
47	Finches	2645
45	Wood-Warblers	2244
60	Crows & Jays	2003
...	...	...
119	Grebes	147
103	Puffbirds	146
4	Ioras	143
70	Albatrosses	143
27	Bustards	141

136 rows × 2 columns

Figure 49 Count the number of images per family.

We can manually verify (some) of these numbers.

```
# Manually check these values for a random family
joined_birds.where(joined_birds["Family Name"] == 'Pheasants').count()
```

```
class id            3583
filepaths           3583
labels              3583
scientific label    3583
data set            3583
Family              3583
Family Name         3583
dtype: int64
```

```
# Manually check these values for a random family
joined_birds.where(joined_birds["Family Name"] == "Bustards").dropna().shape
```

```
(141, 7)
```

```
# Add all the counts up and check that we get our total number of images
sum = 0
for count in family_count["count"]:
    sum += count
print(sum, birds.shape)
```

```
75126 (75126, 6)
```

Figure 50 Verification of family counts.

Then we create a scatter plot that displays the number of images per bird family.

Create a scatter plot to show the distribution of the birds according to family

```
# Prepare data
x = family_count["family name"]
y = family_count["count"]

# Create scatter plot
fig, ax = plt.subplots(figsize=(30,15))
ax.scatter(y=y, x=x)
fig.suptitle('Number of images in the dataset per bird family', fontsize=30)
plt.xticks(rotation = 90)
ax.set_xlim(0,136)
plt.subplots_adjust(bottom=0.26)

# Annotate max and min image count
pheasants = 3583
ax.annotate(text="Pheasants: " + str(pheasants), xy=(0, pheasants), xytext=(1, pheasants), color='r', fontsize=20)
ducks = 3361
ax.annotate(text="Ducks: " + str(ducks), xy=(0, ducks), xytext=(2, ducks), color='r', fontsize=20)
bustards = 141
ax.annotate(text="Bustards: " + str(bustards), xy=(0, bustards), xytext=(136, bustards), color='blue', fontsize=20)

# Display and save plot
plt.savefig('families_scatter_plot.jpeg')
plt.show()
```

Figure 51 Code to create a scatter plot displaying the number of images per bird family.

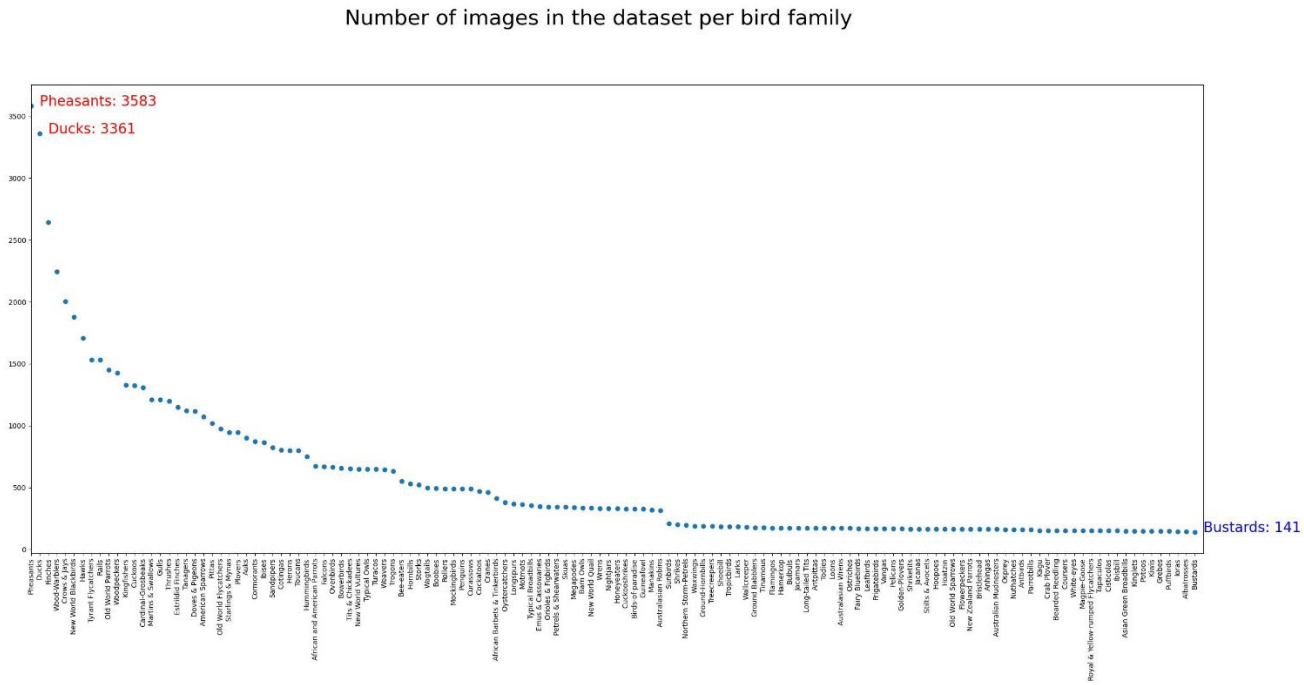


Figure 52 Scatter plot displaying the number of images per bird family.

Visualization 3: 3D Scatter plot of average RGB values

The third visual will be rendered interactively in the user_interface notebook, but because the preparation is done in the visualization notebook, we will discuss it here.

For each image, we calculate the average value of the **R**ed, **G**reen, and **B**lue color channels. We add this to our joined_birds dataframe because this has the families as well, then we remove any unnecessary columns, keeping only the family name and the RGB values. Unfortunately, because we have so many images, the process of getting the average RGB values takes a long time (> 1 hour), so we save the dataframe as a csv file and read it again for further use.

3.Visualize average color

```
# We reuse the joined_birds dataframe, because it already has the families added.

# Use the get_avg_RGB function to add a column that hold the average red, green and blue value of the whole image.
# Note: because we have so many images, this process takes a while. We therefor save the result in a new csv file
# so we can use it later without having to recalculate everything.

joined_birds['rgb']=joined_birds['filepaths'].map(get_avg_RGB)

# Remove unnecessary rows
joined_birds = joined_birds.drop(["Unnamed: 0", "class id", "filepaths", "labels", "scientific label", "data set", "Family" ],
                                 axis=1)

# Save this as a csv file
joined_birds.to_csv('bird_rgb.csv')
```

```
# Read the saved csv file that contains the average rgb values for each image
birds_rgb = pd.read_csv('bird_rgb.csv')
birds_rgb.head()
```

	Family Name	rgb
0	Hawks	[149.12683354591837, 158.53380102040816, 166.5...
1	Hawks	[126.47401147959184, 147.85303730867346, 177.7...
2	Hawks	[75.83113440688776, 77.6499322385204, 63.94015...
3	Hawks	[129.3465202487245, 123.95059390943878, 124.17...
4	Hawks	[87.28698979591837, 80.81184231505102, 75.0546...

Figure 53 Preparation of average RGB colors

```
def get_avg_RGB(path):
    """
    This function returns the average of each band (red, green, blue) of color in an image.
    """
    path = "data/birds/" + path
    image = PIL.Image.open(path)
    average_color = [mean(image.getdata(band)) for band in range(3)]
    image.close()
    return average_color     # Returns a list of 3 values, the average for each color band, as a string
```

Figure 54 The function get_avg_RGB().

We also need a list of unique family names.

```
# Create a list of unique family names
unique_families = birds_rgb["Family Name"].unique()
pd.DataFrame(unique_families).to_csv("unique_families.csv")
unique_families
```

```
array(['Hawks', 'Starlings & Mynas', 'Guineafowl', 'Long-tailed Tits',
       'Ioras', 'Auks', 'Sunbirds', 'Old World Parrots',
       'New World Blackbirds', 'Bowerbirds', 'Ducks', 'Pheasants',
```

Figure 55 Unique family names.

The function colored_scatter takes a family name and displays a 3D scatter plot representing all images belonging to that family. Each dot on the plot has the average color of that image. The x-axis represents green, the y-axis represents red and the z-axis represents blue.

Because the csv file saves the average RGB values as a string and not a list, we use the function string_rgb_to_float() to convert it back to a list of three floats.

```
def colored_scatter(df, family):
    """
    This function displays a 3D scatter plot of the average rgb value of each image within the given family.
    """
    # Keep only the images that belong to the given family and convert the rgb string to floats
    subset = df.where(df["Family Name"] == family).dropna()
    subset = subset.reset_index()
    subset = string_rgb_to_float(subset)

    # Prepare the data
    xs = []  # Red
    ys = []  # Green
    zs = []  # Blue
    for row in range(0,len(subset)):
        xs.append(subset['rgb'][row][0])
        ys.append(subset['rgb'][row][1])
        zs.append(subset['rgb'][row][2])

    # Prepare the list that will color each dot in the 3D scatter plot
    colors = []
    for i in range(0, len(xs)):
        colors.append((xs[i]/255, ys[i]/255, zs[i]/255))

    # Create 3D Scatter plot
    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')
    ax.scatter(xs=xs, ys=ys, zs=zs, color=colors)
    fig.suptitle('Average RGB color for the family ' + family, fontsize=14)
    ax.set_xlabel("Red")
    ax.set_ylabel("Green")
    ax.set_zlabel("Blue")
    ax.xaxis.set_ticklabels([])
    ax.yaxis.set_ticklabels([])
    ax.zaxis.set_ticklabels([]);
```

Figure 56 The function colored_scatter().

```
def string_rgb_to_float(df):
    """
    This function converts RGB values that are in string format to float format in a given dataframe.
    """
    for row in range(0, len(df)):
        df['rgb'][row] = df['rgb'][row].split()            # Splits the string into 3 parts
        df['rgb'][row][0] = float(df['rgb'][row][0][1:-1]) # Red
        df['rgb'][row][1] = float(df['rgb'][row][1][:-1])  # Green
        df['rgb'][row][2] = float(df['rgb'][row][2][:-1])  # Blue
    return df
```

Figure 57 The function string_rgb_to_float().

The result is a 3D scatter plot representing all the images belonging to a given family of birds.

```
# The colored_scatter function creates and displays a 3D scatter plot where each dot represents an image in that family.
colored_scatter(df=birds_rgb, family="Leafbirds")
```

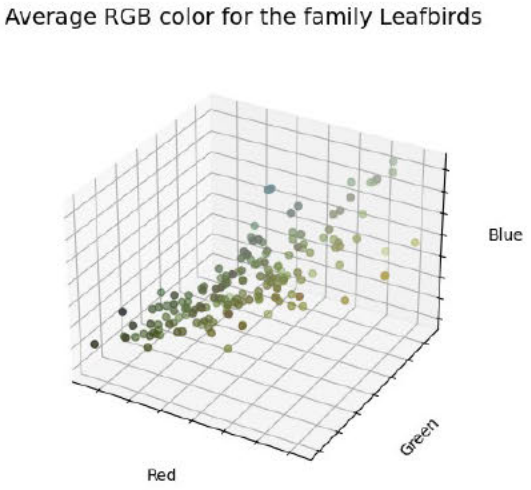


Figure 58 Plot of average RGB values of leafbirds.

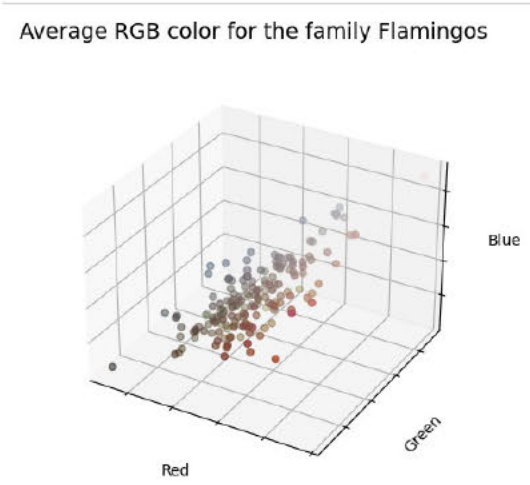


Figure 59 Plot of average RGB values of flamingos.

Making predictions

The last thing we do in this notebook is to make and visualize predictions made by our model. We create our test batches the same way as we did in the model training notebook previously. We do not need the training or validation datasets here.

Because we want to display the image we predicted the label for, we use the function unbatchify() to take the images out of the batches again. The function get_predicted_label() turns the array of probabilities that is returned by the model into the most likely label for the image.

We see that our model predicted that the image was 99% certainly a crested kingfisher, which turns out to be correct when we check the actual label.

```
def unbatchify(batched_data, unique_labels):
    """
    This function takes data in batched form and returns two seperate lists of images and labels.
    If the data does not have any labels, it returns just a list of images.
    """
    images = []
    labels = []

    # Loop through unbatched data and append to lists
    for image, label in batched_data.unbatch().as_numpy_iterator():
        images.append(image)
        if isinstance(label, bytes):
            label = label.decode('utf8')
        else:
            label = unique_labels[np.argmax(label)]
        labels.append(label)
    return images, labels
```

Figure 60 The function unbatchify()

```
def get_predicted_label(predictions, unique_labels):
    """
    This function takes an array of probabilities and returns the highest probable label.
    """
    return unique_labels[np.argmax(predictions)]
```

Figure 61 The function get_predicted_label().

Load model and make predictions on the test data

```
# Load MobileNet model
model = load_model('./models/MobileNet.h5')
```

```
# Make predictions on the test set
test_predictions = model.predict(test_batch, verbose=1)
```

```
71/71 [==============================] - 18s 247ms/step
```

```
# If we want to display the images, we need to pull them out of the batch again
test_images, test_label = unbatchify(batched_data=test_batch, unique_labels=unique_labels)
```

```
# Check if our model predicted the correct label

# Pick a random image
n = 42

# Display the actual image and label
show_num_image(test_images[n], test_label[n])

# Print the predicted label and the probability
print(get_predicted_label(test_predictions[n], unique_labels), test_predictions[n].max())
```

```
CRESTED KINGFISHER 0.9999936
```

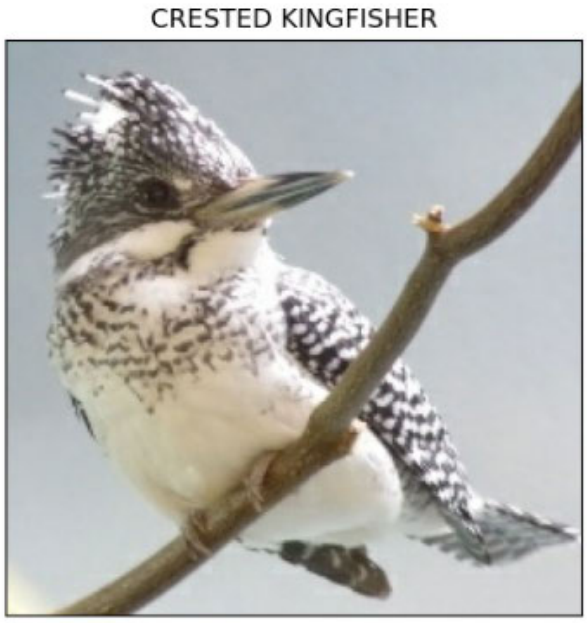


Figure 62 Predicting the label of a random image from the test batch.

We can also predict the labels of images that do not come from the dataset. We downloaded a picture of a goldfinch from the internet and let the model predict the label.

Make predictions on own (user) pictures

```
# Create path to the user picture
path = "./goldfinch.jpg"

# Add the image to a batch (model expects input in the form of batches)
batch = batchify([(path)])

# Let the model predict the label
preds = model.predict(batch)

# Get the predicted label
pred_label = get_predicted_label(preds[0], unique_labels)

# Show the image and the predicted label
show_image(path, pred_label)
```

```
1/1 [==============================] - 1s 663ms/step
```

Figure 63 Make predictions on user images.

C. *User Interface notebook*

The last notebook is primarily used to create the user interface. This is the only notebook that is included in the actual application that is uploaded to Heroku. It relies heavily on the use of widgets and the voila library that can display the notebook as a dashboard.

Because we don't want to display anything other than our dashboard, we do not use markup in this notebook. We also added some code to suppress some warnings.

Setting up the application

```
# User Interface

# This notebook is used to create the user interface for the application.
```

```
# Imports
from numpy import asarray, argmax, sort, where
from pandas import read_csv
from tensorflow_hub import KerasLayer
from io import BytesIO
from warnings import filterwarnings
from IPython.display import display, clear_output
from PIL import Image
import tensorflow as tf
import ipywidgets as widgets
import matplotlib.pyplot as plt
import os

%matplotlib inline

# Disable warnings
filterwarnings('ignore')
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

# !jupyter nbextension enable --py widgetsnbextension --sys-prefix
# !jupyter serverextension enable voila --sys-prefix
```

```
# Constants

# Image input size required by model
IMG_SIZE = 224

# Batch size
BATCH_SIZE = 32

# Path to trained model
MODEL = 'MobileNet.h5'
```

Figure 64 Imports and constants.

We start by defining the layouts for the VBoxes and HBoxes we will use in our application, so they all look uniformly. Then we create the title of the application, as well as the banner that will be displayed at the top, and some instructions for the users.

```
# Set up the application
```

```
# Create layout
layout_vbox = widgets.Layout(display='flex',
                             flex_flow='column',
                             align_items='center',
                             margin='5px 5px 5px 5px',
                             padding='5px 5px 5px 5px')
layout_hbox = widgets.Layout(display='flex',
                             flex_flow='row',
                             align_items='flex-start',
                             justify_content='center',
                             margin='5px 5px 5px 5px',
                             padding='5px 5px 5px 5px')
```

```
# Title and instructions for use
title = widgets.HTML(value="<h1>Bird Classifier</h1>")
instr_1 = widgets.HTML(value="<p>Upload your picture below, then click on the \"Classify\" button to find out" +
                             " the bird in your picture.<p>")
instr_2 = widgets.HTML(value="<p>Or click either the \"Dataset Visuals\" or \"Interactive Visual\" button " +
                             "to explore the dataset on which this application was build.</p>")
```

```
# Banner
file = open("./images/banner.jpg", "rb")
banner_image = file.read()
file.close
banner = widgets.Image(value=banner_image,
                       format='jpg',
                       width='750')
```

Figure 65 Setup of the layout, title, instructions, and banner.

Next, we set up the output box, which is the lower half of the application, where the requested information will be displayed, depending on which buttons the user clicks. We initialize it but leave it empty for now. Likewise, we initialize the box that will hold the result of the interactive third visual, but leave it empty for now.

The uploader is a button that will allow the user to upload their photo to the application. It only accepts the formats ‘jpeg’ and ‘jpg’ to make sure no incompatible files are uploaded to the model.

We read the csv file that holds the unique labels (all 450 bird species) into a list. We also read all the unique families into a list and sort them alphabetically.

Then we load our trained MobileNet model.

```
# Display data visuals as output
outbox = widgets.HBox([], layout=layout_hbox)
visual_3 = widgets.HBox([], layout=layout_hbox)
```

```
# Set up uploader
uploader = widgets.FileUpload(accept='.jpeg, .jpg', multiple=False)
```

```
# Read bird species array
unique_labels = asarray(read_csv("./unique_labels.csv"))

# Read families
all_families = read_csv("./unique_families.csv")
all_families = asarray(all_families)
families = []
for x in all_families:
    families.append(x[1])
families.sort()
```

```
# load model
model = load_model()
```

Figure 66 Setting up the application and loading the data.

The next step is creating buttons that allow the user to choose what he wants to do or display in our application. Each button is linked to a large function that executes when the user clicks the button. We will discuss these later.

```
# Set up 'Classify', 'Dataset Visuals' and 'Interactive Visual' buttons
classify_button = widgets.Button(
                  description='Classify',
                  tooltip='Classify Image',
                  style={'description_width': 'initial'})
visuals_button = widgets.Button(
                  description='Dataset Visuals',
                  tooltip='Display Dataset Characteristics',
                  style={'description_width': 'initial'})
interactive_button = widgets.Button(
                  description='Interactive Visual',
                  tooltip='Display Interactive Visuals',
                  style={'description_width': 'initial'})

hbox_upload = widgets.HBox([uploader, classify_button, visuals_button, interactive_button])

classify_button.on_click(on_classify_button_clicked)
visuals_button.on_click(on_visuals_button_clicked)
interactive_button.on_click(on_interactive_button_clicked)
```

Figure 67 Setting up the user interface buttons.

Next, we group all these parts together in the overall application box, and we create the output widget that will hold and display the output box.

When we execute the “out” widget, our application gets loaded and displayed.

```
# Set up overall application
app_box = widgets.VBox([banner, title, instr_1, instr_2, hbox_upload, outbox], layout=layout_vbox)
```

```
# Set up output field
out = widgets.Output()

with out:
    display(app_box)
```

```
# Display the application
out
```

Bird Classifier

Upload your picture below, then click on the "Classify" button to find out the bird in your picture.

Or click either the "Dataset Visuals" or "Interactive Visual" button to explore the dataset on which this application was build.

Upload (0) | Classify | Dataset Visuals | Interactive Visual

Figure 68 The full application.

Classify Button

The first pair of buttons will allow the user to classify a picture of a bird they upload to the application. When they click the uploader button, they can select a picture they want to upload. Nothing else happens at this time, except that the picture is uploaded to the application.

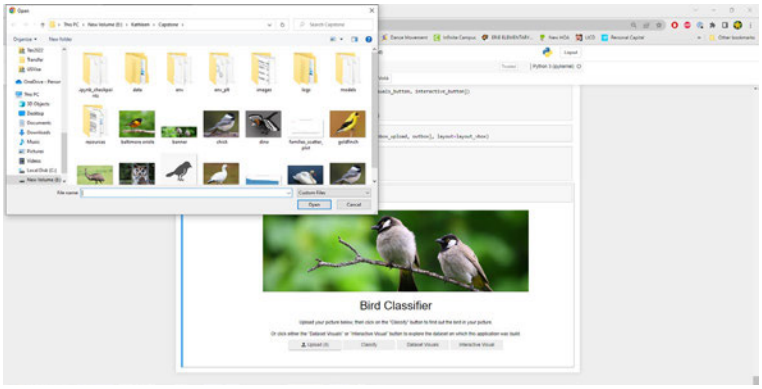


Figure 69 Uploader window.

Next, the user clicks on Classify, which calls the on_classify_button_clicked() function. This is a very large function that calls several other functions.

First, it clears any previous results from the outbox and sets a placeholder where the uploaded image of the bird will later be displayed. It resets the img and img_resized variables.

Then it reads the image from the uploader. At this point, the function checks whether an image was uploaded by checking that img has a value. If the user clicked the “Classify” button before uploading an image, the else clause gets activated, and the user sees the placeholder image with some text that no valid image was uploaded (see figure 74).

If there is an image loaded, the function then resizes it to fit the format that the model is trained on, and saves it so that a path to the image can be fed into the classify() function later. The image is then first converted to a bytes array and then to an Image widget for display later.

As mentioned, the classify() function will then take the image path and run it through the model to get an array of probabilities for each bird species. The get_predicted_label() function we saw in the visualization notebook then retrieves the most likely label for the image.

From these predictions, we also take the top 3 predictions, and their corresponding labels, and turn them into a bar graph.

All these outputs (the user image, the predicted label, and the bar chart with the top 3 predictions) are then combined into a Vbox and loaded into the output window to be displayed (see figure 75).

Lastly, the uploader is reset.

```
def on_classify_button_clicked(event):
    """
    When the 'Classify Image' button is clicked, the uploaded image will be classified by the machine learning model,
    then the uploaded image and a predicted label (bird species) will be displayed.
    """
    # Clear previous result
    global outbox
    global visual_3
    outbox.close()
    visual_3.close()

    # Set placeholder
    outbox = set_placeholder()

    # Reset img and img_resized
    img = None
    img_resized = None

    # Read uploader
    for name, file_info in uploader.value.items():
        # Read image from uploader, resize it and save as jpg
        img = Image.open(BytesIO(file_info['content']))

    # Check that something was uploaded
    if img:
        # Save user image
        img_resized = img.resize((IMG_SIZE,IMG_SIZE))
        img_resized.save("./images/user_image_resized.jpg")

        # Transform original user image to bytes to display later
        img_bytes_array = BytesIO()
        img.save(img_bytes_array, format=img.format)
        img_bytes_array = img_bytes_array.getvalue()

        # Make image and label for output
        output_image = widgets.Image(value=img_bytes_array,
                                     format='jpg',
                                     width=300)
        predictions = classify("./images/user_image_resized.jpg")
        label = widgets.Text(get_predicted_label(predictions, unique_labels))

        # Get the top 3 predictions
        top_3_indeces = (-predictions[0]).argsort()[:3]
        top_3_probs = []
        top_3_labels = []
        for i in top_3_indeces:
            top_3_probs.append(predictions[0][i])
            top_3_labels.append(unique_labels[i][1])

        # Create bar chart of top 3 predictions
        bar_chart = create_bar_chart(top_3_probs, top_3_labels)
        bar_chart_box = widgets.VBox([bar_chart], layout=layout_vbox)

        # Create output
        labeled_bird = widgets.VBox([label, output_image], layout=widgets.Layout(display='flex',
                                                                                  flex_flow='column',
                                                                                  align_items='center',
                                                                                  margin='29px 5px 5px 5px',
                                                                                  padding='5px 5px 5px 5px'))

        outbox = widgets.HBox([labeled_bird, bar_chart_box], layout=layout_hbox)

        # Reset uploader
        uploader.value.clear()
        uploader._counter = 0

        # Display the output
        with out:
            display(outbox)

    else: # No image was uploaded
        outbox = set_placeholder()

    # Reset uploader
        uploader.value.clear()
        uploader.value.clear()
        uploader._counter = 0

    # Display the output
        with out:
            display(outbox)
```

Figure 70 The function on_classify_button_clicked().

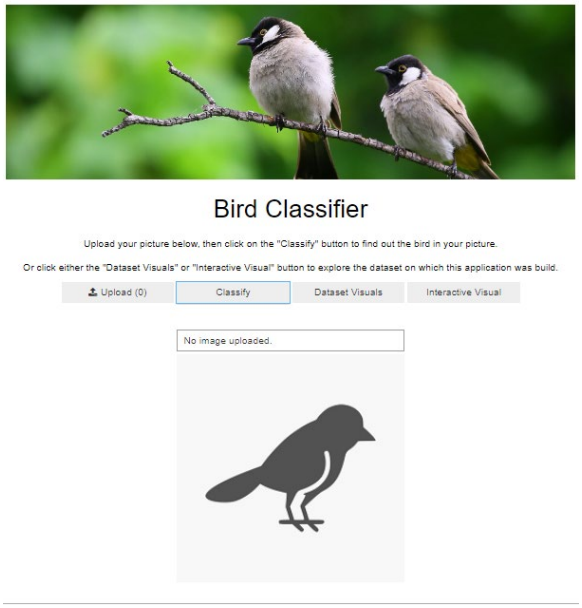


Figure 71 No valid image was uploaded.

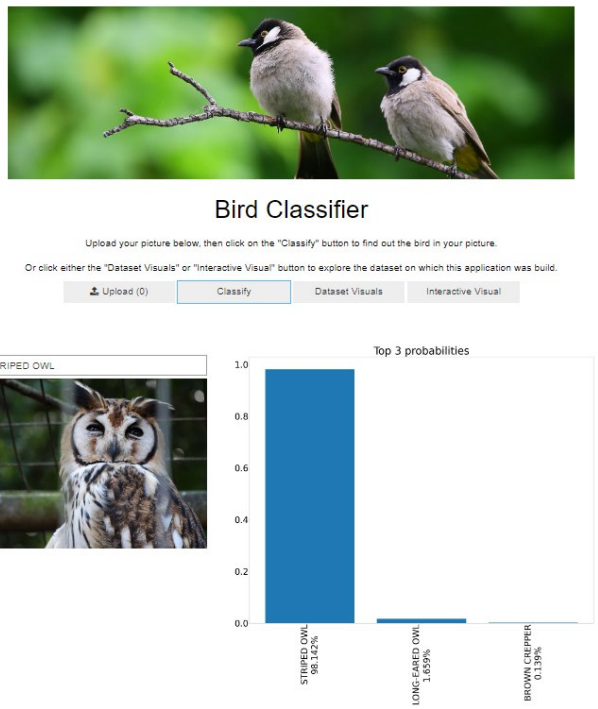


Figure 72 A valid image was uploaded and classified.

The function classify() takes the path to the resized user image and applies the batchify() function to it (see figure 24). Then it lets the model predict the probabilities that the bird in the picture belongs to each of the 450 labels and returns those probabilities.

```
def classify(user_image_path="./images/user_image_resized.jpg"):
    """
    Runs the uploaded image throught the model to predict which bird is pictured.
    """
    # Adds the user image to a batch
    batch = batchify([(user_image_path)])

    # Let the model predict the label
    predictions = model.predict(batch, verbose=0)

    return predictions
```

Figure 73 The function classify().

The create_bar_chart() function takes a list of the top 3 probabilities and the corresponding labels. It concatenates the probabilities in percentage format to the labels so they can be displayed on the x-axis. Then it creates a bar chart that displays the top 3 predicted probabilities for the user image. It saves it, then reopens it, loads it into an Image widget, and returns this widget.

```
def create_bar_chart(probabilities, labels):
    """
    This function takes the probabilities predicted by the model and the corresponding labels and
    returns a bar chart.
    """
    # Add percentages to labels
    labels_w_percentages =  []
    for i, label in enumerate(labels):
        label = label + "\n" + "{:.3f}".format(probabilities[i] * 100) + "%"
        labels_w_percentages.append(label)

    # Make bar chart
    font_size = 50
    fig, ax = plt.subplots(figsize =(30,30))
    ax.bar(labels_w_percentages, probabilities)
    ax.set_title("Top 3 probabilities", fontdict={'fontsize': 60})
    plt.xticks(rotation=90, fontsize=font_size)
    plt.yticks(fontsize=font_size)
    par = {'font.size': font_size,
           'axes.titlesize' : font_size}
    plt.rcParams.update(par)
    plt.tight_layout()

    # Save bar chart
    plt.savefig('./images/bar_chart.jpeg')
    plt.close(fig)

    # Import bar chart image into Image widget
    file = open('./images/bar_chart.jpeg', "rb")
    bar_chart_image = file.read()
    file.close
    bar_chart = widgets.Image(value=bar_chart_image,
                              format='jpg',
                              width = 500)
    return bar_chart
```

Figure 74 The function create_bar_chart().

Dataset Visuals Button

The “Dataset Visuals” button displays the two static visuals that we created in the Visualization notebook: the barplot showing the number of images per species and the scatter plot showing the total number of images per family.

The function that gets triggered when the user clicks the button is called on_visuals_button_clicked(). It starts by clearing out any previous outputs from the output box. Then it reads in the first visual (bar plot). We use the HTML widget to write a formatted explanation of the visual and add the image to an Image widget. The text and the image are then combined into a VBox.

We do the same thing for the second visual. Then we combine the two visuals into the outbox and display it (see figure 79).

```
def on_visuals_button_clicked(event, family = "Ducks"):
    """
    When the 'Visuals' button is clicked, two static visuals pertaining the bird dataset will be displayed. The first is
    a bar plot that displays the number of images per species. The second is a scatter plot that shows the number of
    images per family.
    """
    # Clear previous output
    global outbox
    global visual_3
    outbox.close()
    visual_3.close()

    # Visual 1

    # Read the image
    file = open("./images/species_bar_plot.jpeg", "rb")
    visual_1 = file.read()
    file.close

    # Create the accompanying text and join them together in a VBox
    visual_1_text = widgets.HTML(value="<h2>Number of images per bird species.</h2>" +
                                 "<p>This image shows the number of images that are available for each bird species.</br>" +
                                  "We see that the highest number of images is 258 for the House Finch and the lowest number" +
                                  " is 140 for the Snow Plover. </br></p>" +
                                 "<p>In general, we expect our model to do better at recognizing the bird species that have more
                                  " available to train on. </br> However, since we have more than 100 images for each species, we
                                  " the model to do a decent job on all bird species.</p>")
    visual_1_img = widgets.Image(value=visual_1,
                                 format='jpeg',
                                 width='1000')
    visual_1_vbox = widgets.VBox([visual_1_text, visual_1_img], layout=layout_vbox)

    # Visual 2

    # Read the image
    file = open("./images/families_scatter_plot.jpeg", "rb")
    visual_2 = file.read()
    file.close

    # Create the accompanying text and join them together in a VBox
    visual_2_text = widgets.HTML(value="<h2>Number of images per bird family</h2>" +
                                 "<p>This image grouped the individual species together per family. There are 136 families " +
                                 "in our dataset. </br> We would expect that if the model has many images of birds of the same "
                                 "family (who will likely resemble each other), it will be more adept at recognizing members " +
                                 "of that </br> family." +
                                 " We see that the family with the most images are the pheasants, with 3583 images, closesly fol
                                 "by the ducks with 3361 images. </br> After that the number of images per family drops fast and
                                 "on the right side of the graph we can see that there are many smaller families that have " +
                                 "less than 500 images </br> in them.</p>")
    visual_2_img = widgets.Image(value=visual_2,
                                 format='jpeg',
                                 width='1000')
    visual_2_vbox = widgets.VBox([visual_2_text, visual_2_img], layout=layout_vbox)

    # Join both visuals in the outbox
    outbox = widgets.VBox([visual_1_vbox, visual_2_vbox], layout=layout_vbox)
```

```
    # Display the output
    with out:
        display(outbox)
```

Figure 75 The function on_visuals_button_clicked().

Bird Classifier

Upload your picture below, then click on the "Classify" button to find out the bird in your picture.

Or click either the "Dataset Visuals" or "Interactive Visual" button to explore the dataset on which this application was build.

Upload (0) | Classify | Dataset Visuals | Interactive Visual

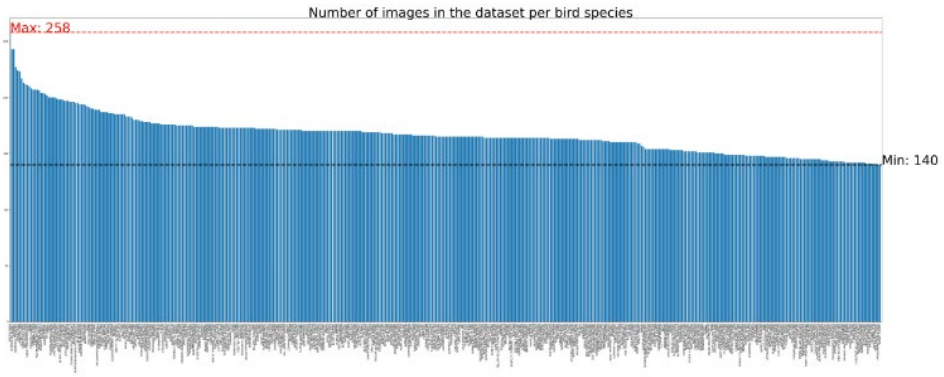
Number of images per bird species.

This image shows the number of images that are available for each bird species.

We see that the highest number of images is 258 for the House Finch and the lowest number is 140 for the Snow Plover.

In general, we expect our model to do better at recognizing the bird species that have more images available to train on.

However, since we have more than 100 images for each species, we expect the model to do a decent job on all bird species.



Number of images per bird family

This image grouped the individual species together per family. There are 136 families in our dataset.

We would expect that if the model has many images of birds of the same family (who will likely resemble each other), it will be more adept at recognizing members of that

family. We see that the family with the most images are the pheasants, with 3583 images, closesly followed by the ducks with 3361 images.

After that the number of images per family drops fast and on the right side of the graph we can see that there are many smaller families that have less than 500 images

in them.

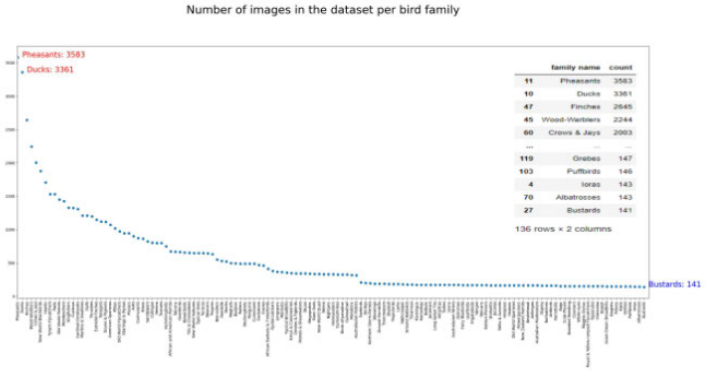


Figure 76 Two static visuals.

Interactive Visual Button

The last button on the dashboard is the “Interactive Visual” Button, which will display the third visual based on the dataset, the average RGB values for each image belonging to a chosen bird family.

The function behind this button is on_interactive_button_clicked(). Just like the other buttons, it starts with clearing any previous outputs. Then it creates a Dropdown widget that allows the user to choose any of the 136 bird families (see figure 81). By default, the widget is set to Ducks.

An interactive widget then passes the selection in the dropdown widget to the family_select() function (see figure 82). This function will create and return the 3D scatter plot, which is then displayed in the outbox.

```
def on_interactive_button_clicked(event):
    """
    When the 'Interactive' button is clicked, an interactive visual is displayed. The user can pick a bird family from the
    dropdown box and a 3D scatter plot will display the average RGB values for each image in that family.
    """
    # Clear previous output
    global outbox
    global visual_3
    outbox.close()
    visual_3.close()

    # Create dropdown
    Dropdown = widgets.Dropdown(options=families, value='Ducks',
                                description='Family:',
                                disabled=False)

    # Display the interactive visual
    with out:
        text = widgets.HTML(value='Select the bird family you would like to display')
        interactive = widgets.interactive(family_select, family = Dropdown, __manual=True)
        outbox = widgets.VBox([text, interactive], layout=layout_vbox)
        display(outbox)
```

Figure 77 The function on_interactive_button_clicked().

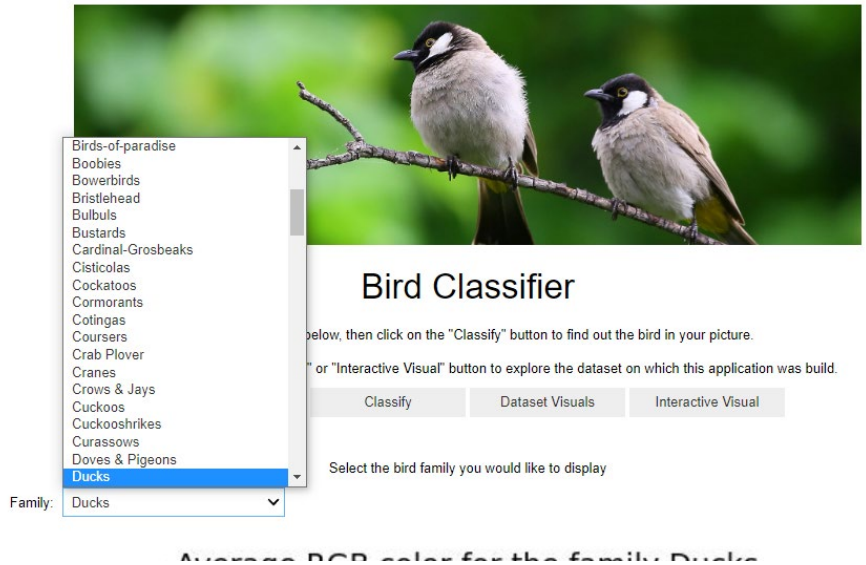


Figure 78 Family dropdown widget.

The family_select() function takes the selected family and calls the colored_scatter() function to create and save the 3D scatter plot for that family. We have discussed the colored_scatter() function in the visualization notebook (see figure 59).

The 3D scatter plot is then read and loaded into an Image widget, which is wrapped into a Vbox and displayed.

```
def family_select(family="Ducks"):
    """
    This function powers the interactive element to display the 3D Scatter plot for the choosen family.
    """
    # Clear previous output
    global visual_3
    visual_3.close()

    # Create and save visual 3
    colored_scatter(family)

    # Open and read the saved image
    file = open("./images/colored_scatter.jpeg", "rb")
    visual_3 = file.read()
    file.close

    # Turn the image into a widget and prepare the VBox
    visual_3_img = widgets.Image(value=visual_3,
                                 format='jpeg',
                                 width='1000')
    visual_3 = widgets.VBox([visual_3_img], layout=layout_vbox)

    # Display the 3D scatter plot
    with out:
        display(visual_3)
```

Figure 79 The family_select() function.

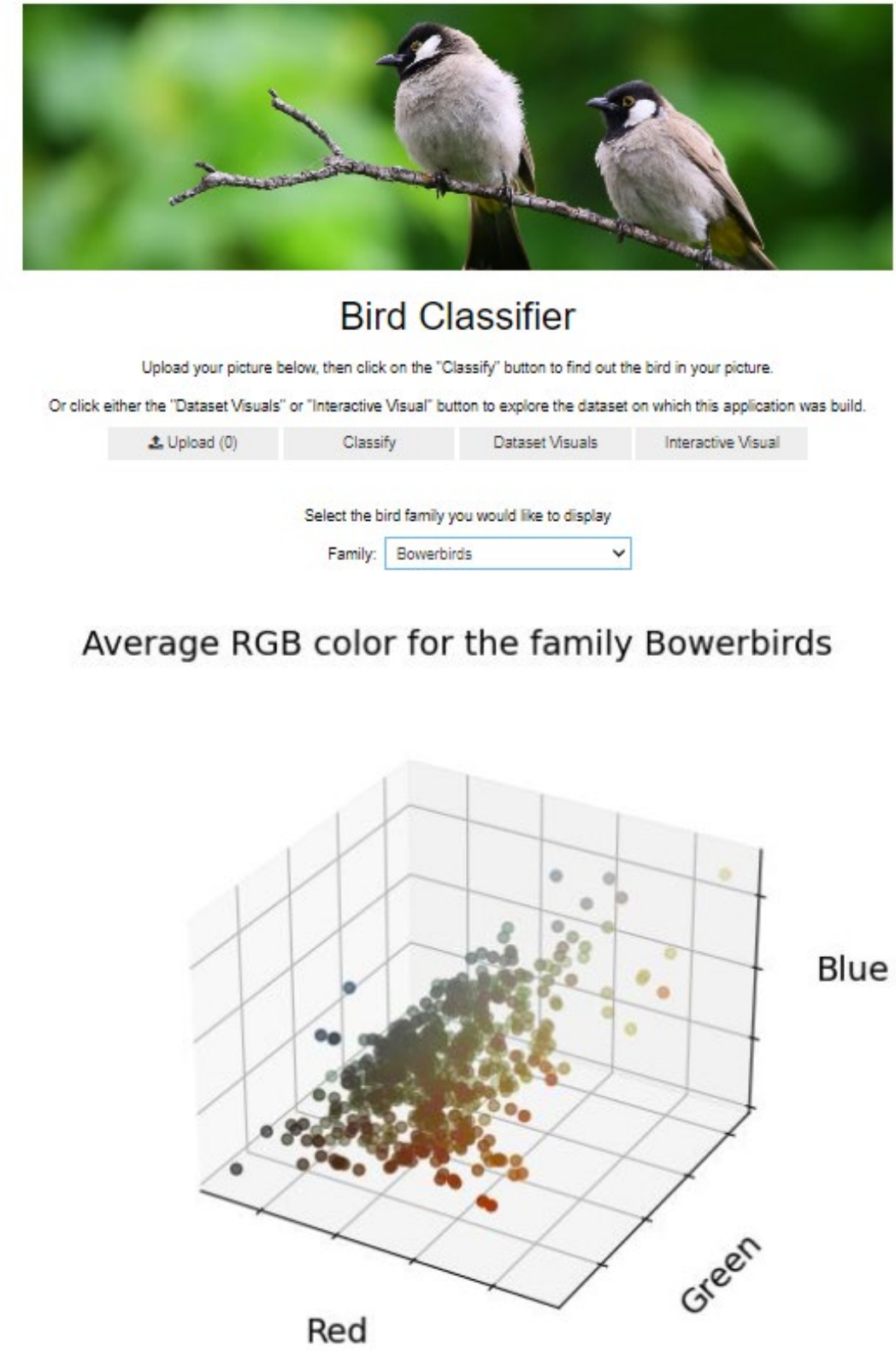


Figure 80 Interactive Visual.

Voila

All the widgets that are used and displayed in this notebook can be displayed as web-application by using the voila library for jupyter notebooks. By clicking the button in the top corner, the notebook is displayed in a separate tab.

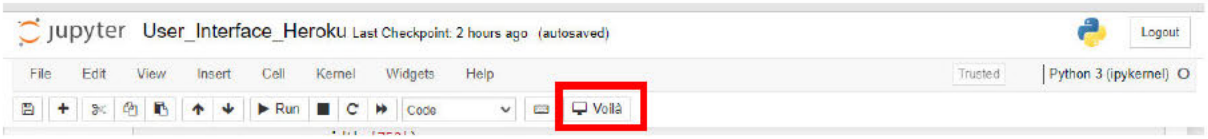


Figure 81 The voila button.

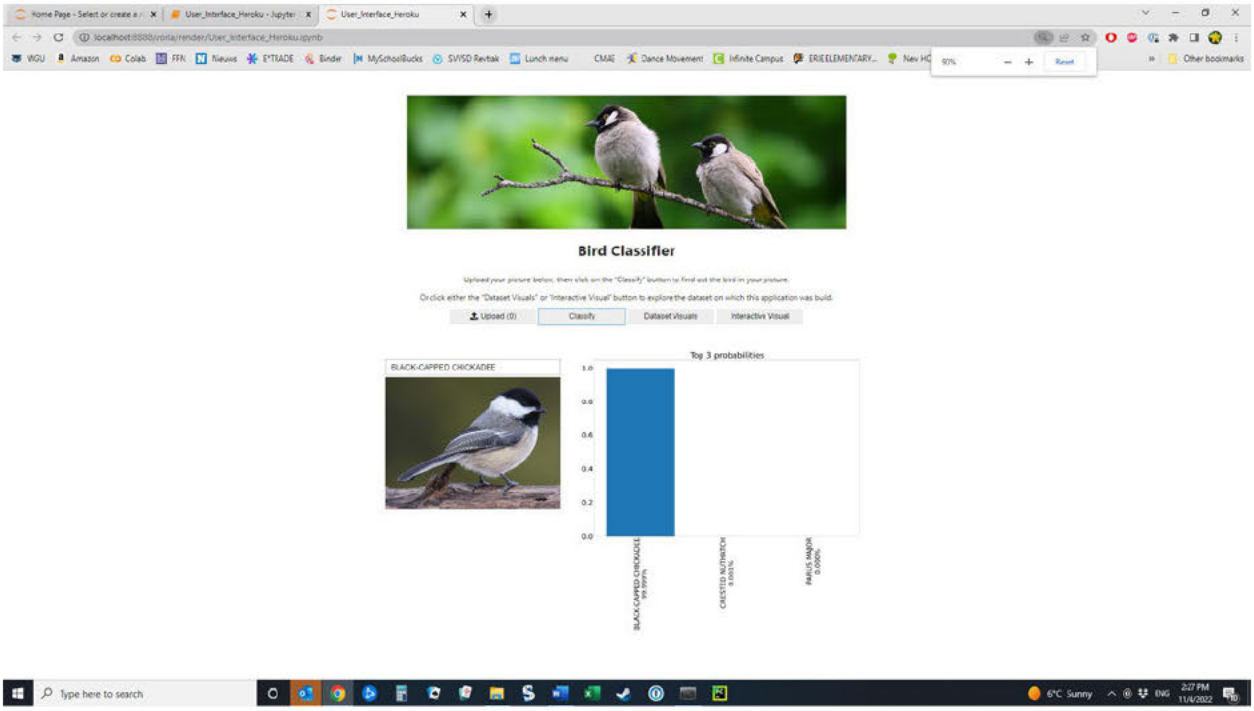


Figure 82 Web-application view.

Heroku

For the application to run on Heroku, we need three configuration files. In the Procfile file, we specify that our application is a web application and that we want voila to automatically open our notebook [redacted] when the application is started so that we see the same dashboard-style application.

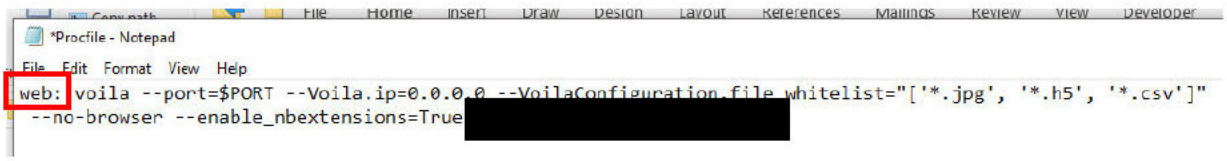


Figure 83 Procfile

The requirements.txt file specifies the libraries needed for Heroku to build the environment in which the application runs. We specify specifically that we want the CPU-only version of TensorFlow (not the GPU-enabled version) to save memory.

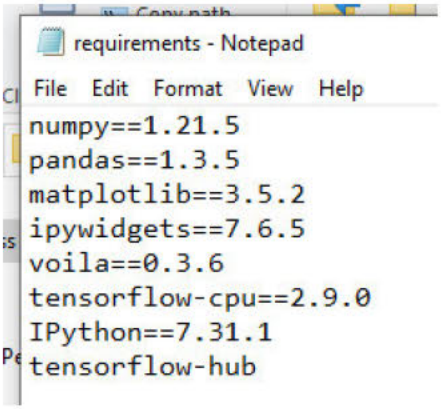


Figure 84 Requirements.txt file.

The runtime.txt file specifies the version of python used for the application.

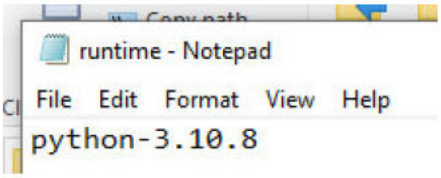


Figure 85 The runtime.txt file.

4. Hypothesis verification

We hypothesized that we can train a CNN model so that it will accurately (> 90%) classify bird images into 450 different species.

As we have seen when we compared the metrics of the three models (ResNet, MobileNet, and EfficientNet), all three manage to have an accuracy of over 90% on the test dataset, so we accept our hypothesis. MobileNet has the highest accuracy, with 95.42%, and is chosen for our application.

	Loss	Accuracy	Precision	Recall
ResNet	0.509602	0.909778	0.915398	0.908889
MobileNet	0.208484	0.954222	0.957942	0.951556
EfficientNet	0.407527	0.928000	0.932499	0.927111

Figure 41 (repeated, cropped) Comparison of the three models.

Our application was meant to be an easily accessible, easy-to-use web application. By the use of a simple interface that can be accessed by the users in their own web browser, we have succeeded in this goal.

5. Effective Visualizations and Reporting

Our notebooks, most notably the Visualizations notebook, contain multiple graphics and visuals that give a better insight into our dataset.

One of the easiest ways of visualizing a dataset containing images is to simply display some of these images, along with their label. We can immediately see that these images are cropped in such a way that the birds are front and center, in close-up. This will help the model recognize the birds better in similar photos, but it might make it harder for the model to recognize birds in pictures where the bird is less prominently displayed. Users might get a better result if they crop their picture to have as many of the pixels belonging to the actual bird.

We can also see that some of these pictures have artifacts in them, like the watermark on the picture of the African Oyster Catcher.

Visualize some of the images

```
#Check some of the images
show_image(filepaths[1000], labels[1000])
```

Figure 44 (repeated) Visualization of a single image.

Figure 45 (repeated) Visualization of multiple images.

Next, we can explore the balance of the different classes (bird species) in the dataset. If a particular species has significantly more images than others, the model might be imbalanced in a way such that it is much more accurate in predicting this species. These species with a lot of images may skew the reported accuracy.

First, we summarize this in table form (abbreviated to only show the head and tail of the dataframe), then we create a bar plot of the number of images per bird species. As we can see, the total number of images in the dataset range between 258 images for the house finch, and 140 images for the Patagonia Sierra finch, the red-tailed thrush, and the snowy plover. However, as we can see in the bar plot, the classes with a lot of images are few in number and most species have more or less the same number of images.

```
# Total (all data sets)
birds["labels"].value_counts()
```

```
HOUSE FINCH                258
D-ARNAUDS BARBET           243
OVENBIRD                   243
SWINHOES PHEASANT          227
WOOD DUCK                  224
                          ...
GO AWAY BIRD               141
BLACK FRANCOLIN            141
RED TAILED THRUSH          140
PATAGONIAN SIERRA FINCH    140
SNOWY PLOVER               140
Name: labels, Length: 450, dtype: int64
```

Figure 49 (repeated). The number of images in the whole dataset.

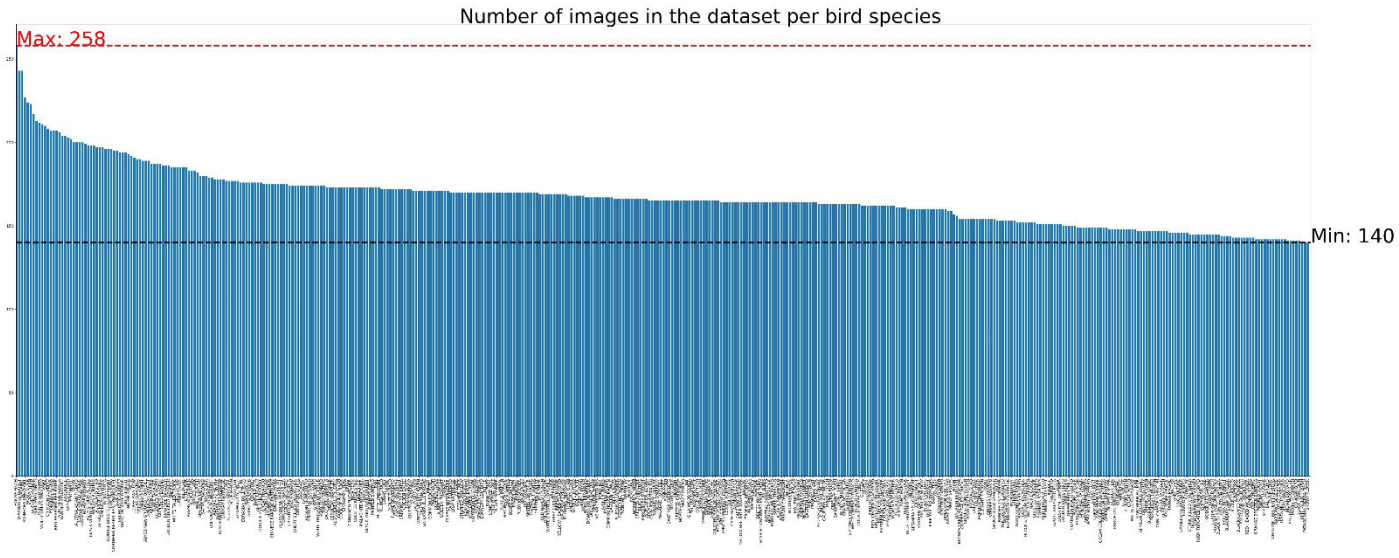


Figure 51 (repeated) Bar plot of images per species.

Another way of looking at this is by grouping the species per family. The idea is that the model will most likely confuse a species with other members of their family since birds belonging to the same family are likely to have similar appearances.

Again, we can first summarize this in a dataframe, from which we can see that the most populous families are the pheasants and the ducks, each with more than 3000 images. Some of the families with much fewer images are the albatrosses and the bustards. It is worth noting that one of the reasons for this could be that there simply are more species in the pheasant family than in the bustard family. Still, we expect our model to be able to recognize families with more images better than those with fewer images.

When we look at these numbers in a scatter plot, we can see some families with a very large number of images, and a long tail of families with relatively fewer images.

	family name	count
11	Pheasants	3583
10	Ducks	3361
47	Finches	2645
45	Wood-Warblers	2244
60	Crows & Jays	2003
...	...	...
119	Grebes	147
103	Puffbirds	146
4	Ioras	143
70	Albatrosses	143
27	Bustards	141

136 rows × 2 columns

Figure 52 (repeated, cropped) Count the number of images per family.

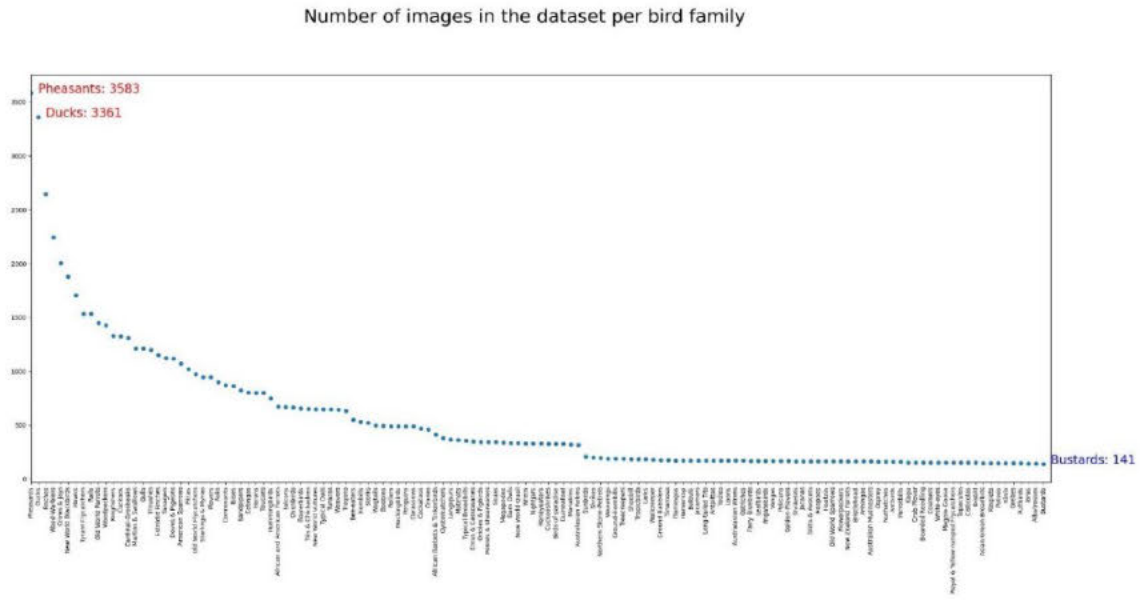
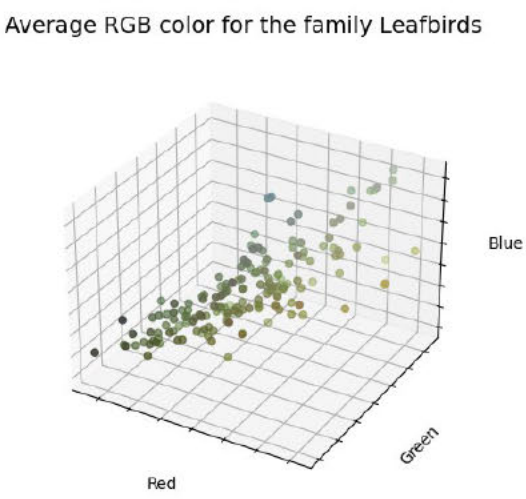


Figure 55 (repeated) Scatter plot displaying the number of images per bird family.

One of the ways bird species differ from one another is their coloration and this is likely an important feature on which our model classifies images. We can visualize this diversity in coloration by taking the average values of the RGB values for each pixel in the image. Color in images is defined by 3 color values for each pixel, which together make up one color. For example, a pixel with an RGB value of RGB(0,128,128) means that the pixel has a red value of 0 (out of a max of 255), a green value of 128, and a blue value of 128. Together, that gives the color teal.

For each image, we took the average of the RGB values (separate per color channel) over all the pixels, then we plotted each image of a certain family on a 3D scatter plot, so we can see the variation of color within the images of that family as well as differences between families.

For example, we can see that the leafbirds are greener, while the flamingos are reddish. The points in the leafbirds plot are more scattered, while the ones in the flamingo family are more clustered together.



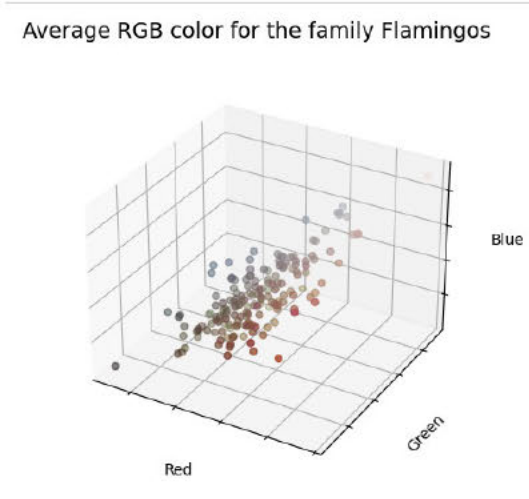


Figure 61 (repeated) Plot of average RGB values of leafbirds.

Figure 62 (repeated) Plot of average RGB values of flamingos.

When the application classifies a user image, it shows the image uploaded by the user, the predicted label, and a bar plot of the top 3 probabilities for its label. This can help the user determine how sure the model is in determining the species.

For example, the model is pretty sure this is a striped owl, but it's not sure about this Archeopetryx at all.

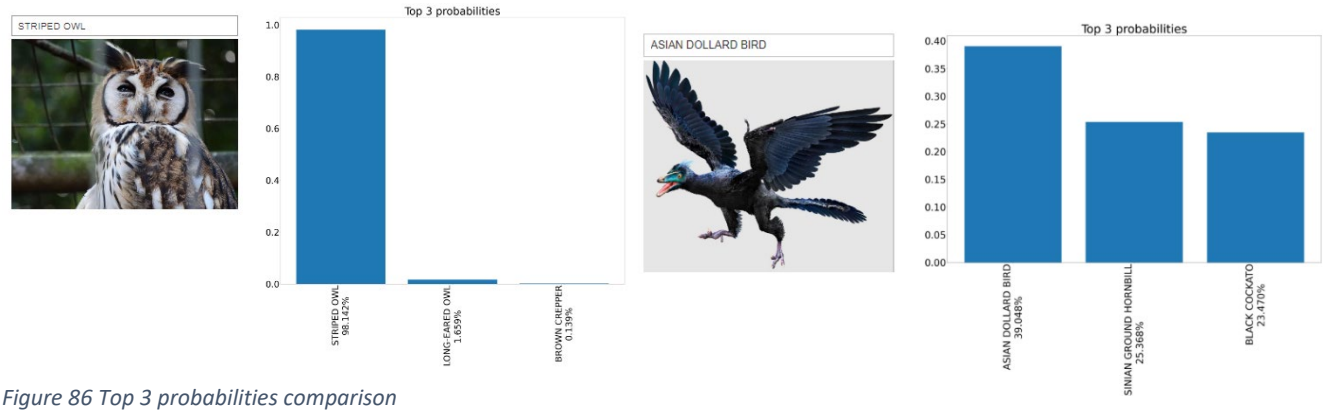


Figure 86 Top 3 probabilities comparison

6. Accuracy analysis

The accuracy of the model was assessed in two ways. First, we looked at the accuracy of the model on the validation data. We can do this by viewing the TensorBoard, which shows the accuracy of the model after each epoch. We see that the MobileNet accuracy reaches 93% after 8 epochs.

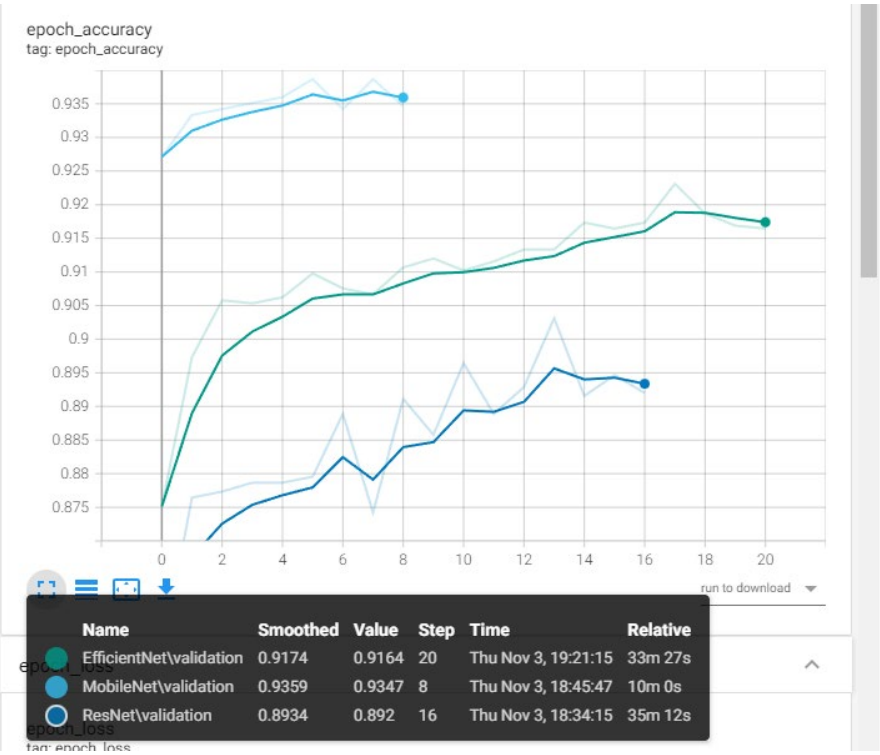


Figure 33 (repeated, cropped) Accuracy for the three models.

While we are mainly interested in the model's accuracy, it is a good idea to look at some of the other metrics as well.

Our model has a loss of 0.27 at the end of the training, which was better than both other models.

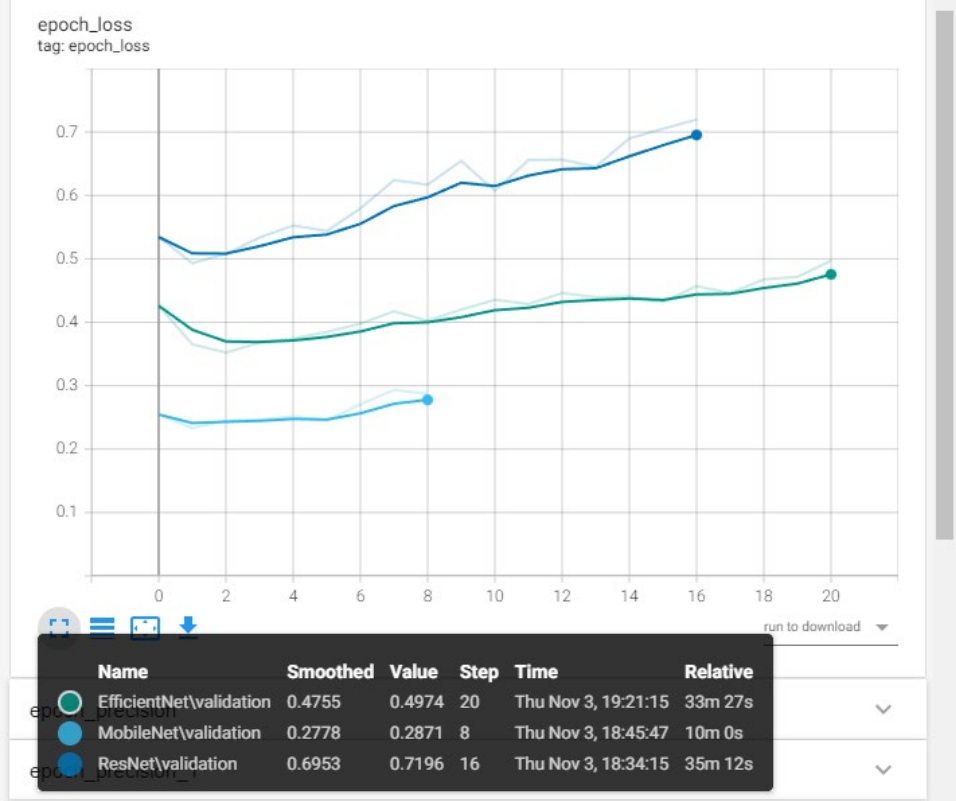


Figure 34 (repeated) Loss for the three models.

MobileNet has a final precision of 94%, and a recall of 93%, both of which are relatively good, and better than the scores of the other two models we tried.

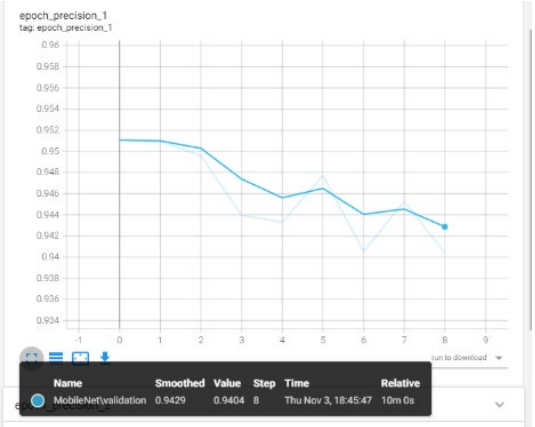


Figure 36 (repeated) Precision of MobileNet.

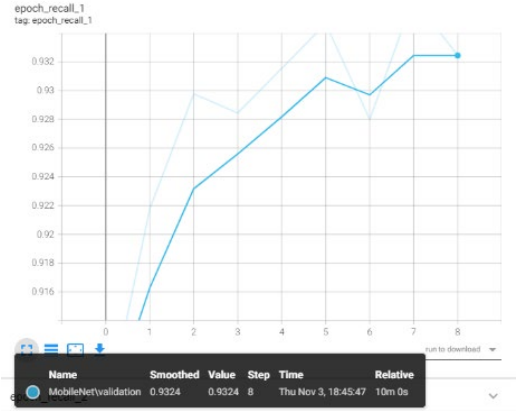


Figure 39 (repeated) Recall of MobileNet.

Secondly, we can evaluate the accuracy of the model on the test dataset as well. We see that our model does relatively well here too. It reaches an accuracy of 95.42%, a loss of 0.21, a precision of 95.79%, and a recall of 95.15%. Again, the MobileNet performed best of all three models.

	Loss	Accuracy	Precision	Recall
ResNet	0.509602	0.909778	0.915398	0.908889
MobileNet	0.208484	0.954222	0.957942	0.951556
EfficientNet	0.407527	0.928000	0.932499	0.927111

Figure 41 (repeated, cropped) Comparison of the three models.

7. Application Testing

We used three different levels of testing for this application: Unit testing, Integration testing, and Acceptance testing.

Acceptance testing was performed by the end customer, [redacted] They determined that the application met their goals and requirements.

We did unit testing on each part of the application:

- Classifying functionality
- Static visuals
- Interactive visual
- Overall application (title, banner, buttons)

Unit testing encompassed going through the code line by line, ensuring no inconsistencies or mistakes were present in the code. Then we tested the functionality by trying out different user cases - including situations where we expect our model to fail - such as feeding the model a correctly formatted image, and an incorrectly formatted image, not feeding it any image at all, feeding it an image of something entirely different, etc.

One problem we encountered while unit testing was that the application crashed if no image was uploaded before the user clicked the "Classify" button. Since the uploader widget did not throw an exception when trying to access an uploaded image when none was uploaded, we had to manually check whether an image was present or not. This also included resetting the image at the start of the function.

```
# Reset img and img_resized
img = None
img_resized = None

# Read uploader
for name, file_info in uploader.value.items():
    # Read image from uploader, resize it and save as jpg
    img = Image.open(BytesIO(file_info['content']))

# Check that something was uploaded
if img:
    # Save user image
    img_resized = img.resize((IMG_SIZE,IMG_SIZE))
    img_resized.save("./images/user_image_resized.jpg")
```

Figure 87 Manually check for the uploaded image.

We also performed integration testing to assure that all units integrate well together. For example, we realized that we needed to clear out the outbox manually every time we went from one button to another, otherwise the previous output would still be visual underneath the new output.

```
    # Clear previous output
    global outbox
    global visual_3
    outbox.close()
    visual_3.close()
```

Figure 88 Clearing previous output.

Output of "Display Visuals"

Output of "Interactive Visual"

Figure 89 Old output persists when changing buttons.

8. Security Measures and Maintenance

This application does not warrant a user login or encryption of data. No sensitive or private information is stored or provided by the application. To protect the application from attacks, we specify a specific format of files that are whitelisted by the application. This is done in the Procfile, needed by Heroku. Any formats other than the ones specified will not be trusted by the application.

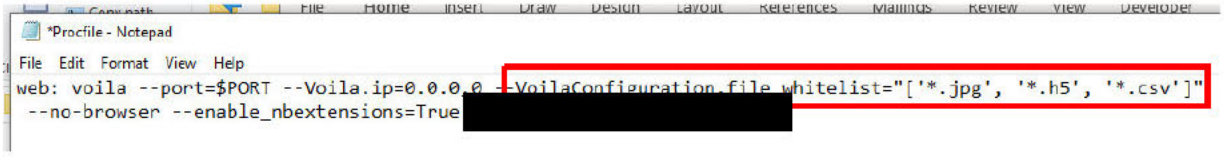


Figure 86 (repeated) Procfile

Heroku, as a PaaS service, also provides a long list of security measures for their systems, protecting their own systems as well as the applications running on them. These measures can be viewed here: https://www.heroku.com/policy/security (Heroku, 2022).

Heroku also provides logs, which allow us to check up on the application and any problems it might encounter.

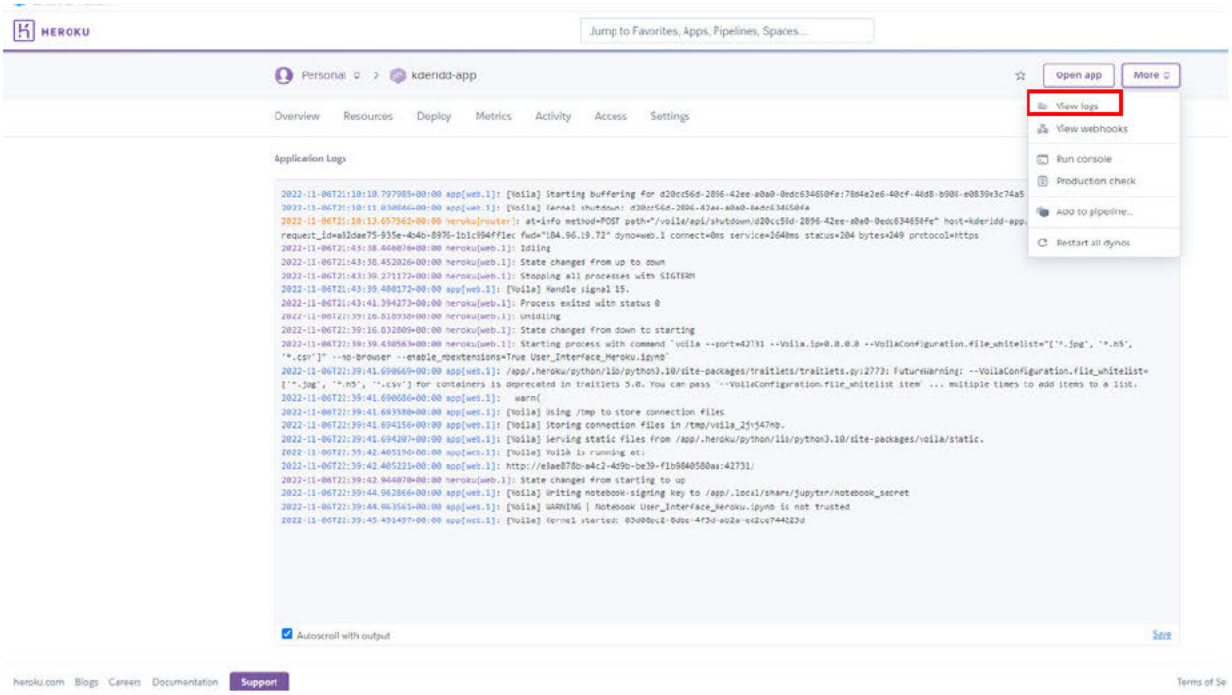


Figure 90 Heroku logs.

9. Application Files

The project includes the following files:

Name	Description
Application documents	These documents are uploaded to Heroku as part of the application
[redacted]	The jupyter notebook in which the application is built.
MobileNet.h5	The trained model
bird_rgb.csv	A CSV list of the average RGB values of all images and the family they belong to.
unique_labels.csv	CSV file listing all 450 bird species
unique_families.csv	CSV file listing all 136 bird families
Procfile	Configuration file for Heroku
requirements.txt	Environment requirements needed to build the application on Heroku
runtime.txt	Specifies the python version the application runs on
Folder "images"	Contains all images needed by the application
banner.jpg	Banner on top of the application
bar_chart.jpg	Top 3 probabilities bar chart (gets overwritten)
colored_scatter.jpg	3D scatter chart (gets overwritten)

species_bar_plot.jpg	Bar plot of the number of images per bird species (visual 1)
families_scatter_plot.jpg	Scatter plot of the number of images per bird family (visual 2)
placeholder.jpg	Placeholder image for when no valid image was uploaded
user_image_resized.jpg	The image that holds the user image for display (gets overwritten)
Supporting documents	These documents are not uploaded as part of the application but were used in the development of the application.
birds.CSV	CSV file that was included in the dataset, listing all images with their filepaths, labels, scientific labels, and which dataset they belong to (train, test, validation)
bird_species_list.CSV	CSV file containing all bird families, with their common name, the genera and the species that belong to that family, and their common names.
Model training.ipynb	Jupyter notebook that was used to train the model(s).
Visualization.ipynb	Jupyter notebook that was used to visualize the data
Folder "logs"	The folder holding the logs for model training. This is used by the tensorboard to visualize the training process and the associated metrics. It has a subfolder for each training event (here: for each of the three models), each containing subfolders for training metrics and validation metrics.
Folder "models"	The folder holding the trained models.
EfficientNet.h5	Trained EfficientNet model
(MobileNet.h5)	Trained MobileNet model, used in (and moved to) application documents
ResNet.h5	Trained ResNet model
Folder "test images"	Folder containing several images to test the application
Environments*	These files can be used to recreate the environments used to develop the application. They are in yml format.
env.yml	The environment used for the Model training notebook
env_plt.yml	The environment used for the User Interface and the Visualizations notebooks. Can also be used to open the Model training notebook, if no actual model training will be done.

*: Heroku builds the required environment for the application to run automatically, based on the requirements.txt document. The two provided yml files are **not necessary** to run the application, but they might be useful to open the notebooks used for development if desired. They can be used by running the conda command:

conda env create -f env.yml

Note: Because the dataset is too large (>2GB), we did not include it here. The dataset can be accessed at [redacted] (Piosenka, 2022). We did include the bird.csv file that accompanied the dataset and has all filepaths.

10. User's Guide

The application is hosted online at the following address: [redacted] No installation is needed.

1. Click [redacted] or paste this address in the browser address bar: [redacted]
   Because the application is quite large and we use the free version, it is possible that on rare occasions, Heroku hangs. Simply refresh the page or click the link again.

2. To classify an image of a bird:
   a. Have an image of a bird you like to classify ready on your computer. It will need to be in jpg or jpeg format.
   b. Click the "Upload" button.
   c. Browse to the image on your computer. For the sake of convenience, several testing images were provided in the submission, but any image of a bird will do.
   d. Click "Open".
   e. Click "Classify".
3. To see the static visuals:
   a. Click "Dataset Visuals".
   b. Scroll down to see the second visual.
4. To see the interactive visual:
   a. Click "Interactive Visual".
   b. In the dropdown menu, select a family you would like the view.

11. Summation of Learning Experience

This project was my first hands-on machine learning and web application development experience, having no professional experience in the field of computer science yet. Therefore, I was largely unfamiliar with virtually all the tools required for it, such as Jupyter notebooks, NumPy, pandas, matplotlib, TensorFlow, voila, and Heroku. However, many resources for self-teaching are available online, such as video courses, user guides, and documentation, and I managed to acquire the necessary knowledge to successfully finish the project successfully. I am confident that the knowledge I gained as well as the skill in knowing how to look for and where to find the needed information, will be very useful in my later career.

Section E

Citations

Boyd, J. H. (2019, July 27). *Taxonomy in flux checklist 3.08*. Taxonomy in Flux Checklist: Version 3.08. Retrieved November 6, 2022, from http://jboyd.net/Taxo/List.html

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. *arXiv*. https://doi.org/10.48550/arXiv.1512.03385

Heroku. (2022). *Heroku security*. Heroku. Retrieved November 6, 2022, from https://www.heroku.com/policy/security

Heroku. (2022). *Platform as a Service*. Heroku. Retrieved November 6, 2022, from https://www.heroku.com/platform

Howard, A., Sandler, M., Chu, G., Chen, L., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., Le, Q. V., & Adam, H. (2019). Searching for MobileNetV3. *arXiv*. https://doi.org/10.48550/arXiv.1905.02244

Piosenka, G. (Updated 2022, October). Kaggle: Birds 450 species - Image classification. Retrieved November 6, 2022, from [redacted]

Russakovsky, O., Deng, J., Su, H. et al. (2015). ImageNet Large Scale Visual Recognition Challenge. *Int J Comput Vis* **115**, 211–252 (2015). https://doi.org/10.1007/s11263-015-0816-y

Tan, M., & Le, Q. V. (2021). EfficientNetV2: Smaller Models and Faster Training. *arXiv*. https://doi.org/10.48550/arXiv.2104.00298