Wildlife Detection System: Enhanced Notebook Structure

Directory Structure

Create the following directory structure in your project:

```
/WildlifeDetectionSystem/notebooks/
├── 01_data_preparation/
│   ├── data_exploration.ipynb
│   ├── dataset_preparation.ipynb
│   └── yolo_export.ipynb
│
├── 02_model_training/
│   ├── model_selection.ipynb
│   ├── hierarchical_training.ipynb
│   └── species_finetuning.ipynb
│
├── 03_evaluation/
│   ├── performance_evaluation.ipynb
│   ├── threshold_analysis.ipynb
│   └── error_analysis.ipynb
│
└── 04_dashboard_integration/
    ├── metrics_generation.ipynb
    └── dashboard_preview.ipynb
```

Notebook Details

01_data_preparation/data_exploration.ipynb

Purpose: Analyze and understand your wildlife dataset

Key Sections:

1. **Load and Count Images**

python

```python
import os
import pandas as pd
from pathlib import Path

# Define paths
raw_data_dir = "/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/data/raw_images"

# Get image counts by directory
image_counts = {}
for root, dirs, files in os.walk(raw_data_dir):
    rel_path = os.path.relpath(root, raw_data_dir)
    if rel_path == '.':
        rel_path = 'root'
    image_files = [f for f in files if f.lower().endswith(('.jpg', '.jpeg', '.png')
    if image_files:
        image_counts[rel_path] = len(image_files)

# Display image distribution
counts_df = pd.DataFrame(list(image_counts.items()), columns=['Directory', 'Image C
counts_df = counts_df.sort_values('Image Count', ascending=False)
counts_df
```

2. **Extract EXIF and Image Metadata**

python

```python
from PIL import Image, ExifTags
import random

def extract_image_metadata(image_path):
    try:
        img = Image.open(image_path)
        metadata = {
            "size": img.size,
            "format": img.format,
            "mode": img.mode
        }

        # Extract EXIF data if available
        if hasattr(img, '_getexif') and img._getexif():
            exif = {
                ExifTags.TAGS[k]: v
                for k, v in img._getexif().items()
                if k in ExifTags.TAGS and not isinstance(v, bytes)
            }
            metadata["exif"] = exif

        return metadata
    except Exception as e:
        return {"error": str(e)}

# Sample random images
all_images = []
for root, _, files in os.walk(raw_data_dir):
    for file in files:
        if file.lower().endswith(('.jpg', '.jpeg', '.png')):
            all_images.append(os.path.join(root, file))

# Sample 20 random images
sample_images = random.sample(all_images, min(20, len(all_images)))

# Extract metadata
metadata_samples = {img: extract_image_metadata(img) for img in sample_images}
```

3. **Analyze YOLO Annotations**

```
python
```

```python
# Define path to existing YOLO dataset
yolo_dataset_path = "/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/data/export

# Read class names
with open(os.path.join(yolo_dataset_path, 'classes.txt'), 'r') as f:
    class_names = [line.strip() for line in f.readlines()]

# Count annotations by class
train_labels_dir = os.path.join(yolo_dataset_path, 'labels/train')
val_labels_dir = os.path.join(yolo_dataset_path, 'labels/val')

class_counts = {i: {'train': 0, 'val': 0} for i in range(len(class_names))}

# Process training files
for label_file in os.listdir(train_labels_dir):
    with open(os.path.join(train_labels_dir, label_file), 'r') as f:
        for line in f.readlines():
            parts = line.strip().split()
            if len(parts) >= 5:
                class_id = int(parts[0])
                if class_id < len(class_names):
                    class_counts[class_id]['train'] += 1

# Process validation files
for label_file in os.listdir(val_labels_dir):
    with open(os.path.join(val_labels_dir, label_file), 'r') as f:
        for line in f.readlines():
            parts = line.strip().split()
            if len(parts) >= 5:
                class_id = int(parts[0])
                if class_id < len(class_names):
                    class_counts[class_id]['val'] += 1

# Display class distribution
class_distribution = []
for class_id, counts in class_counts.items():
    if counts['train'] > 0 or counts['val'] > 0:
        total = counts['train'] + counts['val']
        train_pct = (counts['train'] / total * 100) if total > 0 else 0
        class_distribution.append({
            'Class ID': class_id,
            'Class Name': class_names[class_id],
            'Train Count': counts['train'],
            'Val Count': counts['val'],
            'Total': total,
            'Train %': train_pct,
```

```
            'Val %': 100 - train_pct
        })

pd.DataFrame(class_distribution).sort_values('Total', ascending=False)
```

4. **Visualize Class Distribution**

```python
import matplotlib.pyplot as plt
import seaborn as sns

# Prepare data for visualization
vis_data = pd.DataFrame(class_distribution)

# Plot class distribution
plt.figure(figsize=(12, 8))
sns.barplot(x='Class Name', y='Total', data=vis_data.sort_values('Total', ascending
plt.xticks(rotation=45, ha='right')
plt.title('Distribution of Top 15 Classes in Dataset')
plt.tight_layout()
plt.show()

# Plot train/val split
plt.figure(figsize=(12, 8))
vis_data_top = vis_data.sort_values('Total', ascending=False).head(15)

x = range(len(vis_data_top))
plt.bar(x, vis_data_top['Train Count'], label='Train')
plt.bar(x, vis_data_top['Val Count'], bottom=vis_data_top['Train Count'], label='Va

plt.xlabel('Class')
plt.ylabel('Number of Annotations')
plt.title('Train/Validation Split for Top 15 Classes')
plt.xticks(x, vis_data_top['Class Name'], rotation=45, ha='right')
plt.legend()
plt.tight_layout()
plt.show()
```

01_data_preparation/dataset_preparation.ipynb

Purpose: Prepare balanced datasets with improved annotations

Key Sections:

1. **Define Hierarchical Categories**

python

```python
# Define taxonomic groups for hierarchical classification
taxonomic_groups = {
    'Deer': [0, 1, 2, 3],  # Red Deer, Male Roe Deer, Female Roe Deer, Fallow Deer
    'Carnivores': [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16],  # Fox, Wolf, Jackal, e
    'Small_Mammals': [17, 18, 19, 20, 21],  # Rabbit, Hare, Squirrel, etc.
    'Birds': [23, 24, 25, 29],  # Blackbird, Nightingale, Pheasant, woodpecker
    'Other': [4, 5, 22, 26, 27, 28]  # Wild Boar, Chamois, Turtle, Human, Backgroun
}

# Create inverse mapping from class_id to group
class_to_group = {}
for group_name, class_ids in taxonomic_groups.items():
    for class_id in class_ids:
        class_to_group[class_id] = group_name
```

2. **Identify Underrepresented Classes**

python

```python
# Find underrepresented classes
underrepresented = []
for record in class_distribution:
    if record['Total'] < 5:  # Consider classes with fewer than 5 examples as under
        underrepresented.append({
            'Class Name': record['Class Name'],
            'Class ID': record['Class ID'],
            'Total': record['Total'],
            'Taxonomic Group': class_to_group.get(record['Class ID'], 'Unknown')
        })

pd.DataFrame(underrepresented).sort_values('Total')
```

3. **Implement Data Augmentation Strategy**

```python
import albumentations as A
import cv2
import numpy as np

# Define stronger augmentation for rare classes
strong_aug = A.Compose([
    A.RandomRotate90(),
    A.Flip(),
    A.RandomBrightnessContrast(brightness_limit=0.3, contrast_limit=0.3),
    A.HueSaturationValue(hue_shift_limit=20, sat_shift_limit=30, val_shift_limit=20
    A.GaussNoise(),
    A.Perspective(),
    A.ShiftScaleRotate(shift_limit=0.2, scale_limit=0.2, rotate_limit=30)
])

# Define standard augmentation for common classes
standard_aug = A.Compose([
    A.RandomRotate90(),
    A.Flip(),
    A.RandomBrightnessContrast(brightness_limit=0.2, contrast_limit=0.2)
])

def augment_yolo_sample(image_path, label_path, augmentation, output_image_path, ou
    """Augment a YOLO format image and its labels"""
    # Implementation details
```

4. **Balance Dataset**

```python
# Function to balance dataset by augmenting underrepresented classes
def balance_dataset(yolo_dataset_path, output_path, target_count=10):
    """
    Balance dataset by augmenting underrepresented classes to reach target_count
    """
    # Implementation details
```

01_data_preparation/yolo_export.ipynb

Purpose: Export prepared data to YOLO format for training

Key Sections:

1. **Configure Export Settings**

python

```python
import os
import yaml
import shutil
from datetime import datetime
from pathlib import Path

# Define export paths
timestamp = datetime.now().strftime("%Y%m%d_%H%M")
export_dir = f"/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/data/export/yolo_

# Create export directories
os.makedirs(os.path.join(export_dir, 'images/train'), exist_ok=True)
os.makedirs(os.path.join(export_dir, 'images/val'), exist_ok=True)
os.makedirs(os.path.join(export_dir, 'labels/train'), exist_ok=True)
os.makedirs(os.path.join(export_dir, 'labels/val'), exist_ok=True)
```

2. **Create Dataset YAML Configuration**

python

```python
# Create data.yaml file
data_yaml = {
    'path': export_dir,
    'train': 'images/train',
    'val': 'images/val',
    'nc': len(class_names),
    'names': class_names
}

# Write YAML file
with open(os.path.join(export_dir, 'data.yaml'), 'w') as f:
    yaml.dump(data_yaml, f, sort_keys=False)

print(f"Created data.yaml with {len(class_names)} classes")
```

3. **Export Two Dataset Variants**

```python
# Export standard dataset (species-level classification)
def export_species_dataset(source_path, export_dir):
    """Export dataset with original species labels"""
    # Implementation details

# Export hierarchical dataset (taxonomic group classification)
def export_hierarchical_dataset(source_path, export_dir, taxonomic_groups):
    """Export dataset with taxonomic group labels"""
    # Implementation details
```

02_model_training/model_selection.ipynb

Purpose: Select optimal model architecture based on hardware capabilities

Key Sections:

1. **Detect Hardware Capabilities**

```python
import torch
import platform
import psutil

# Check system capabilities
def get_system_info():
    """Get system hardware information"""
    system_info = {
        "platform": platform.platform(),
        "processor": platform.processor(),
        "python_version": platform.python_version(),
        "ram_gb": round(psutil.virtual_memory().total / (1024**3), 2)
    }

    # Check CUDA availability
    cuda_available = torch.cuda.is_available()
    system_info["cuda_available"] = cuda_available

    if cuda_available:
        system_info["cuda_version"] = torch.version.cuda
        system_info["gpu_name"] = torch.cuda.get_device_name(0)
        system_info["gpu_count"] = torch.cuda.device_count()

        # Get available GPU memory
        torch.cuda.empty_cache()
        gpu_memory_gb = torch.cuda.get_device_properties(0).total_memory / (1024**3
        system_info["gpu_memory_gb"] = round(gpu_memory_gb, 2)

    return system_info

system_info = get_system_info()
system_info
```

2. **Determine Optimal Model Configuration**

```
python
```

```python
def get_optimal_model_config(system_info):
    """
    Determine optimal model configuration based on hardware

    Returns:
        dict: Configuration including model type, image size, batch size
    """
    if not system_info["cuda_available"]:
        # CPU-only configuration
        return {
            "model_type": "yolov8n.pt",  # Nano model for CPU
            "image_size": 320,
            "batch_size": 1,
            "workers": 0,
            "device": "cpu"
        }

    # GPU configurations
    gpu_memory = system_info.get("gpu_memory_gb", 0)

    if gpu_memory >= 12:
        # High-end GPU
        return {
            "model_type": "yolov8l.pt",  # Large model
            "image_size": 640,
            "batch_size": 16,
            "workers": 4,
            "device": 0  # First GPU
        }
    elif gpu_memory >= 8:
        # Mid-range GPU
        return {
            "model_type": "yolov8m.pt",  # Medium model
            "image_size": 640,
            "batch_size": 8,
            "workers": 4,
            "device": 0
        }
    elif gpu_memory >= 4:
        # Entry-level GPU
        return {
            "model_type": "yolov8s.pt",  # Small model
            "image_size": 416,
            "batch_size": 8,
            "workers": 2,
            "device": 0
```

```
        }
    else:
        # Low-memory GPU
        return {
            "model_type": "yolov8n.pt",  # Nano model
            "image_size": 320,
            "batch_size": 4,
            "workers": 2,
            "device": 0
        }

model_config = get_optimal_model_config(system_info)
model_config
```

3. **Configure Memory Optimization Strategies**

python

```python
def configure_memory_optimization(system_info, model_config):
    """Add memory optimization strategies to model configuration"""
    optimization = {}

    # Start with the base model config
    optimization.update(model_config)

    # CPU-specific optimizations
    if not system_info["cuda_available"]:
        optimization["cache"] = "disk"  # Use disk cache instead of RAM
        optimization["half"] = False    # Don't use half precision on CPU
        return optimization

    # GPU-specific optimizations
    gpu_memory = system_info.get("gpu_memory_gb", 0)

    # Always use half precision on GPU when possible
    optimization["half"] = True

    # Enable gradient accumulation for smaller GPUs to simulate larger batches
    if gpu_memory < 8:
        nominal_batch_size = 16
        actual_batch_size = optimization["batch_size"]
        optimization["nbs"] = nominal_batch_size  # Nominal batch size
        print(f"Using gradient accumulation: {nominal_batch_size} effective batch s

    # Very low memory - force CPU for some operations
    if gpu_memory < 2:
        optimization["cache"] = "disk"

    return optimization

optimized_config = configure_memory_optimization(system_info, model_config)
optimized_config
```

4. **Create Complete Training Configuration**

```
python
```

```
import json
from datetime import datetime

# Get timestamp for model name
timestamp = datetime.now().strftime("%Y%m%d_%H%M")

# Locate most recent YOLO dataset
def find_latest_dataset():
    """Find most recent YOLO dataset in export directory"""
    export_dir = "/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/data/export"
    yolo_dirs = [d for d in os.listdir(export_dir) if d.startswith("yolo_") and os.

    if not yolo_dirs:
        return None

    # Sort directories by creation time (newest first)
    latest_dir = sorted(yolo_dirs, key=lambda d: os.path.getmtime(os.path.join(expo
    return os.path.join(export_dir, latest_dir)

latest_dataset = find_latest_dataset()
data_yaml_path = os.path.join(latest_dataset, "data.yaml") if latest_dataset else N

# Create complete training configuration
training_config = {
    # Model and data
    "model_type": optimized_config["model_type"],
    "data": data_yaml_path,
    "imgsz": optimized_config["image_size"],

    # Output settings
    "project": "/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/models/trained",
    "name": f"wildlife_detector_{timestamp}",
    "save": True,
    "save_period": 10,

    # Training parameters
    "epochs": 100,
    "patience": 25,
    "batch": optimized_config["batch_size"],
    "workers": optimized_config.get("workers", 0),
    "device": optimized_config.get("device", "cpu"),

    # Optimization parameters
    "optimizer": "AdamW",
    "lr0": 0.001,
    "lrf": 0.01,
```

```
    "momentum": 0.937,
    "weight_decay": 0.0005,
    "warmup_epochs": 5,
    "warmup_momentum": 0.8,
    "warmup_bias_lr": 0.1,

    # Loss function weights
    "box": 7.5,    # Box loss gain
    "cls": 3.0,    # Class loss gain
    "dfl": 1.5,    # Distribution focal loss gain

    # Data augmentation
    "hsv_h": 0.015,  # HSV Hue augmentation
    "hsv_s": 0.7,    # HSV Saturation augmentation
    "hsv_v": 0.4,    # HSV Value augmentation
    "degrees": 10.0,  # Rotation augmentation
    "translate": 0.2, # Translation augmentation
    "scale": 0.6,     # Scale augmentation
    "fliplr": 0.5,    # Horizontal flip probability
    "mosaic": 1.0,    # Mosaic augmentation
    "mixup": 0.1      # Mixup augmentation
}

# Add memory optimization parameters
if "half" in optimized_config:
    training_config["half"] = optimized_config["half"]

if "nbs" in optimized_config:
    training_config["nbs"] = optimized_config["nbs"]

if "cache" in optimized_config:
    training_config["cache"] = optimized_config["cache"]

# Save configuration to file
os.makedirs("/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/models/config", exi
config_path = f"/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/models/config/tr

with open(config_path, 'w') as f:
    json.dump(training_config, f, indent=2)

print(f"Training configuration saved to: {config_path}")
```

02_model_training/hierarchical_training.ipynb

Purpose: Train model using hierarchical approach with taxonomic groups

Key Sections:

1. **Configure Hierarchical Training**

```python
import os
import json
import yaml
from ultralytics import YOLO

# Find latest configuration
config_dir = "/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/models/config"
config_files = [f for f in os.listdir(config_dir) if f.startswith("training_config_
latest_config_file = sorted(config_files, reverse=True)[0]
config_path = os.path.join(config_dir, latest_config_file)

# Load configuration
with open(config_path, 'r') as f:
    config = json.load(f)

# Find hierarchical dataset
data_dir = "/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/data/export"
hierarchical_dirs = [d for d in os.listdir(data_dir) if d.startswith("yolo_hierarch

if hierarchical_dirs:
    # Use most recent hierarchical dataset
    latest_dir = sorted(hierarchical_dirs, reverse=True)[0]
    hierarchical_dataset = os.path.join(data_dir, latest_dir)
    hierarchical_yaml = os.path.join(hierarchical_dataset, "data.yaml")

    # Update configuration to use hierarchical dataset
    config["data"] = hierarchical_yaml
    config["name"] = f"{config['name']}_hierarchical"
else:
    print("No hierarchical dataset found. Will create one now.")
    # Code to create hierarchical dataset would go here
```

2. **Initialize and Train Model**

python

```python
# Initialize YOLOv8 model
model = YOLO(config["model_type"])

# Display training configuration
print(f"Training {config['model_type']} model on hierarchical data")
print(f"Image size: {config['imgsz']}")
print(f"Batch size: {config['batch']}")
print(f"Device: {config['device']}")

# Start training
results = model.train(**config)

# Save training summary
model_dir = os.path.join(config["project"], config["name"])
summary_path = os.path.join(model_dir, "training_summary.json")

summary = {
    "map50": float(results.maps[1]),
    "map50-95": float(results.maps[0]),
    "epochs_completed": int(results.epoch),
    "best_epoch": results.best_epoch,
    "training_time_seconds": results.t[-1] + results.epoch * results.t[0]
}

with open(summary_path, 'w') as f:
    json.dump(summary, f, indent=2)

print(f"Training complete! Results saved to {model_dir}")
print(f"Best mAP50: {summary['map50']:.4f}, Best epoch: {summary['best_epoch']}")
```

3. **Visualize Training Results**

```python
import matplotlib.pyplot as plt
import pandas as pd

# Load training results
results_csv = os.path.join(model_dir, "results.csv")
results_df = pd.read_csv(results_csv)

# Find metric columns
precision_col = next((col for col in results_df.columns if 'precision' in col.lower
recall_col = next((col for col in results_df.columns if 'recall' in col.lower()), N
map50_col = next((col for col in results_df.columns if 'map50' in col.lower() and '
map50_95_col = next((col for col in results_df.columns if 'map50-95' in col.lower()

# Plot training metrics
plt.figure(figsize=(12, 8))

if precision_col:
    plt.plot(results_df['epoch'], results_df[precision_col], label='Precision')
if recall_col:
    plt.plot(results_df['epoch'], results_df[recall_col], label='Recall')
if map50_col:
    plt.plot(results_df['epoch'], results_df[map50_col], label='mAP50')
if map50_95_col:
    plt.plot(results_df['epoch'], results_df[map50_95_col], label='mAP50-95')

plt.xlabel('Epoch')
plt.ylabel('Metric Value')
plt.title('Training Metrics Over Time')
plt.legend()
plt.grid(True)
plt.tight_layout()

# Save the plot
plt.savefig(os.path.join(model_dir, "training_metrics.png"))
plt.show()
```

02_model_training/species_finetuning.ipynb

Purpose: Fine-tune hierarchical model for individual species

Key Sections:

1. **Load Hierarchical Model**

python

```python
import os
import json
from ultralytics import YOLO

# Find latest hierarchical model
models_dir = "/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/models/trained"
hierarchical_models = [d for d in os.listdir(models_dir) if d.endswith("_hierarchic

if not hierarchical_models:
    print("Error: No hierarchical models found. Train a hierarchical model first.")
else:
    # Use most recent hierarchical model
    latest_model = sorted(hierarchical_models, reverse=True)[0]
    model_dir = os.path.join(models_dir, latest_model)
    model_path = os.path.join(model_dir, "weights/best.pt")

    # Load the model
    if os.path.exists(model_path):
        model = YOLO(model_path)
        print(f"Loaded hierarchical model: {model_path}")
    else:
        print(f"Error: Model weights not found at {model_path}")
```

2. **Configure Fine-tuning**

```python
# Find original species dataset
data_dir = "/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/data/export"
species_dirs = [d for d in os.listdir(data_dir) if d.startswith("yolo_") and not d.

if species_dirs:
    # Use most recent species dataset
    latest_dir = sorted(species_dirs, reverse=True)[0]
    species_dataset = os.path.join(data_dir, latest_dir)
    species_yaml = os.path.join(species_dataset, "data.yaml")

    # Load original configuration and modify for fine-tuning
    with open(os.path.join(model_dir, "args.yaml"), 'r') as f:
        config = yaml.safe_load(f)

    # Update configuration for fine-tuning
    timestamp = datetime.now().strftime("%Y%m%d_%H%M")
    finetuning_config = {
        # Use original dataset with species labels
        "data": species_yaml,

        # Output settings
        "project": "/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/models/train
        "name": f"wildlife_detector_{timestamp}_finetuned",
        "save": True,

        # Training parameters - fewer epochs for fine-tuning
        "epochs": 50,
        "patience": 15,

        # Lower learning rate for fine-tuning
        "lr0": 0.0005,  # Lower initial learning rate

        # Keep other parameters from original training
        "imgsz": config.get("imgsz", 640),
        "batch": config.get("batch", 16),
        "device": config.get("device", 0),
        "workers": config.get("workers", 4)
    }
```

3. **Fine-tune the Model**

python

```python
# Start fine-tuning
print(f"Fine-tuning model on species data")
print(f"Base model: {model_path}")
print(f"Dataset: {species_yaml}")
print(f"Learning rate: {finetuning_config['lr0']}")

results = model.train(**finetuning_config)

# Save training summary
finetuned_dir = os.path.join(finetuning_config["project"], finetuning_config["name"
summary_path = os.path.join(finetuned_dir, "finetuning_summary.json")

summary = {
    "base_model": model_path,
    "map50": float(results.maps[1]),
    "map50-95": float(results.maps[0]),
    "epochs_completed": int(results.epoch),
    "best_epoch": results.best_epoch,
    "training_time_seconds": results.t[-1] + results.epoch * results.t[0]
}

with open(summary_path, 'w') as f:
    json.dump(summary, f, indent=2)

print(f"Fine-tuning complete! Results saved to {finetuned_dir}")
print(f"Best mAP50: {summary['map50']:.4f}, Best epoch: {summary['best_epoch']}")
```

4. **Compare with Base Model**

```python
# Compare performance with hierarchical model
base_results_csv = os.path.join(model_dir, "results.csv")
finetuned_results_csv = os.path.join(finetuned_dir, "results.csv")

base_df = pd.read_csv(base_results_csv)
finetuned_df = pd.read_csv(finetuned_results_csv)

# Find metric columns
base_map50_col = next((col for col in base_df.columns if 'map50' in col.lower() and
finetuned_map50_col = next((col for col in finetuned_df.columns if 'map50' in col.l

if base_map50_col and finetuned_map50_col:
    base_best_map50 = base_df[base_map50_col].max()
    finetuned_best_map50 = finetuned_df[finetuned_map50_col].max()

    improvement = (finetuned_best_map50 - base_best_map50) * 100

    print(f"Base model best mAP50: {base_best_map50:.4f}")
    print(f"Fine-tuned model best mAP50: {finetuned_best_map50:.4f}")
    print(f"Improvement: {improvement:.2f}%")
```

03_evaluation/performance_evaluation.ipynb

Purpose: Comprehensively evaluate model performance

Key Sections:

1. **Load Model and Configure Evaluation**

```python
import os
import json
import yaml
from ultralytics import YOLO
import pandas as pd
from datetime import datetime

# Find latest model (either hierarchical or fine-tuned)
models_dir = "/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/models/trained"
model_dirs = [d for d in os.listdir(models_dir) if os.path.isdir(os.path.join(model

if not model_dirs:
    print("Error: No models found.")
else:
    # Use most recent model
    latest_model = sorted(model_dirs, reverse=True)[0]
    model_dir = os.path.join(models_dir, latest_model)
    model_path = os.path.join(model_dir, "weights/best.pt")

    # Load the model
    if os.path.exists(model_path):
        model = YOLO(model_path)
        print(f"Loaded model: {model_path}")
    else:
        print(f"Error: Model weights not found at {model_path}")

# Find dataset used for training
with open(os.path.join(model_dir, "args.yaml"), 'r') as f:
    training_args = yaml.safe_load(f)

data_yaml = training_args.get("data")
if not data_yaml or not os.path.exists(data_yaml):
    print(f"Error: Data YAML not found at {data_yaml}")
else:
    print(f"Using dataset: {data_yaml}")

# Create evaluation output directory
timestamp = datetime.now().strftime("%Y%m%d_%H%M")
eval_dir = f"/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/reports/evaluation_
os.makedirs(eval_dir, exist_ok=True)
```

2. **Run Comprehensive Evaluation**

python

```python
# Configure evaluation parameters
eval_config = {
    "data": data_yaml,
    "batch": 16,
    "imgsz": training_args.get("imgsz", 640),
    "conf": 0.25,  # Default confidence threshold
    "iou": 0.7,    # IoU threshold for NMS
    "max_det": 300,  # Maximum detections per image
    "task": "val",
    "device": training_args.get("device", 0),
    "verbose": True,
    "save_json": True,  # Save results in COCO JSON format
    "save_hybrid": True,  # Save hybrid version of labels
    "save_conf": True    # Save confidences in .txt labels
}

# Run validation
print("\nRunning comprehensive evaluation...")
results = model.val(**eval_config)

# Extract key metrics
metrics = {
    "mAP50": float(results.box.map50),
    "mAP50-95": float(results.box.map),
    "precision": float(results.box.mp),
    "recall": float(results.box.mr)
}

print("\nOverall Performance Metrics:")
print(f"- mAP50: {metrics['mAP50']:.4f}")
print(f"- mAP50-95: {metrics['mAP50-95']:.4f}")
print(f"- Precision: {metrics['precision']:.4f}")
print(f"- Recall: {metrics['recall']:.4f}")
```

3. **Extract Per-class Metrics**

python

```python
# Extract and format per-class metrics
per_class_metrics = {}

if hasattr(results, 'detailed_results'):
    for i, class_result in enumerate(results.detailed_results):
        class_name = results.names[i]
        per_class_metrics[class_name] = {
            "precision": float(class_result.get("precision", 0)),
            "recall": float(class_result.get("recall", 0)),
            "map50": float(class_result.get("map50", 0)),
            "map50-95": float(class_result.get("map", 0))
        }
else:
    # Try to extract from another source
    with open(os.path.join(model_dir, "val_results.txt"), 'r') as f:
        lines = f.readlines()

    for line in lines:
        if line.startswith("  ") and not line.startswith("   all"):
            parts = line.strip().split()
            if len(parts) >= 7:
                class_name = " ".join(parts[:-6])
                precision = float(parts[-6])
                recall = float(parts[-5])
                map50 = float(parts[-4])
                map50_95 = float(parts[-3])

                per_class_metrics[class_name] = {
                    "precision": precision,
                    "recall": recall,
                    "map50": map50,
                    "map50-95": map50_95
                }

# Print per-class metrics for top classes
print("\nPer-class Performance (Top 5 by mAP50):")
top_classes = sorted(per_class_metrics.items(), key=lambda x: x[1]["map50"], revers

for class_name, metrics in top_classes:
    print(f"- {class_name}:")
    print(f"  Precision: {metrics['precision']:.4f}")
    print(f"  Recall: {metrics['recall']:.4f}")
    print(f"  mAP50: {metrics['map50']:.4f}")
```

4. **Generate and Save Metrics**

```
python
```

```
# Get training history
results_csv = os.path.join(model_dir, "results.csv")
if os.path.exists(results_csv):
    results_df = pd.read_csv(results_csv)

    # Find metric columns
    precision_col = next((col for col in results_df.columns if 'precision' in col.l
    recall_col = next((col for col in results_df.columns if 'recall' in col.lower()
    map50_col = next((col for col in results_df.columns if 'map50' in col.lower() a
    map50_95_col = next((col for col in results_df.columns if 'map50-95' in col.low

    # Create training history
    training_history = {
        "epoch": results_df["epoch"].tolist(),
        "precision": results_df[precision_col].tolist() if precision_col else [],
        "recall": results_df[recall_col].tolist() if recall_col else [],
        "mAP50": results_df[map50_col].tolist() if map50_col else [],
        "mAP50-95": results_df[map50_95_col].tolist() if map50_95_col else []
    }

    # Determine best epoch
    if map50_col:
        best_epoch_idx = results_df[map50_col].idxmax()
        best_epoch = int(results_df.loc[best_epoch_idx, "epoch"])
    else:
        best_epoch = 0
else:
    training_history = {"epoch": []}
    best_epoch = 0

# Create comprehensive metrics
performance_metrics = {
    "precision": metrics["precision"],
    "recall": metrics["recall"],
    "mAP50": metrics["mAP50"],
    "mAP50-95": metrics["mAP50-95"],
    "training_epochs": max(training_history["epoch"]) if training_history["epoch"]
    "best_epoch": best_epoch,
    "classes": len(per_class_metrics),
    "per_class": per_class_metrics,
    "history": training_history
}

# Save metrics files
metrics_file = os.path.join(eval_dir, "performance_metrics.json")
with open(metrics_file, 'w') as f:
```

```
    json.dump(performance_metrics, f, indent=2)

class_metrics_file = os.path.join(eval_dir, "class_metrics.json")
with open(class_metrics_file, 'w') as f:
    json.dump(per_class_metrics, f, indent=2)

history_file = os.path.join(eval_dir, "training_history.json")
with open(history_file, 'w') as f:
    json.dump(training_history, f, indent=2)

print(f"\nMetrics saved to {eval_dir}")
```

03_evaluation/threshold_analysis.ipynb

Purpose: Analyze model performance across different confidence thresholds

Key Sections:

1. **Load Model and Configure Thresholds**

```python
import os
import json
import yaml
from ultralytics import YOLO
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime

# Find latest model
models_dir = "/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/models/trained"
model_dirs = [d for d in os.listdir(models_dir) if os.path.isdir(os.path.join(model

if not model_dirs:
    print("Error: No models found.")
else:
    # Use most recent model
    latest_model = sorted(model_dirs, reverse=True)[0]
    model_dir = os.path.join(models_dir, latest_model)
    model_path = os.path.join(model_dir, "weights/best.pt")

    # Load the model
    if os.path.exists(model_path):
        model = YOLO(model_path)
        print(f"Loaded model: {model_path}")
    else:
        print(f"Error: Model weights not found at {model_path}")

# Find dataset used for training
with open(os.path.join(model_dir, "args.yaml"), 'r') as f:
    training_args = yaml.safe_load(f)

data_yaml = training_args.get("data")

# Create evaluation output directory
timestamp = datetime.now().strftime("%Y%m%d_%H%M")
eval_dir = f"/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/reports/evaluation_
os.makedirs(eval_dir, exist_ok=True)

# Define confidence thresholds to evaluate
thresholds = [0.05, 0.1, 0.25, 0.5, 0.75, 0.9]
```

2. **Evaluate Model Across Thresholds**

python

```python
# Configure base evaluation parameters
base_config = {
    "data": data_yaml,
    "batch": 16,
    "imgsz": training_args.get("imgsz", 640),
    "iou": 0.7,
    "max_det": 300,
    "task": "val",
    "device": training_args.get("device", 0),
    "verbose": False
}

# Evaluate across thresholds
threshold_results = []

print(f"Evaluating model across {len(thresholds)} thresholds...")

for threshold in thresholds:
    print(f"Testing confidence threshold: {threshold:.2f}")

    # Set threshold
    eval_config = base_config.copy()
    eval_config["conf"] = threshold

    # Run validation
    results = model.val(**eval_config)

    # Extract metrics
    metrics = {
        "threshold": threshold,
        "precision": float(results.box.mp),
        "recall": float(results.box.mr),
        "mAP50": float(results.box.map50),
        "mAP50-95": float(results.box.map),
        "f1": 2 * results.box.mp * results.box.mr / (results.box.mp + results.box.m
    }

    threshold_results.append(metrics)
    print(f"  mAP50: {metrics['mAP50']:.4f}, Precision: {metrics['precision']:.4f},

# Create DataFrame for analysis
threshold_df = pd.DataFrame(threshold_results)
threshold_df = threshold_df.sort_values("threshold")
```

3. **Visualize Threshold Effects**

```python
# Plot precision-recall curve across thresholds
plt.figure(figsize=(12, 8))
plt.plot(threshold_df["recall"], threshold_df["precision"], 'o-', linewidth=2, mark

# Label points with thresholds
for i, row in threshold_df.iterrows():
    plt.annotate(f"{row['threshold']:.2f}",
                 (row["recall"], row["precision"]),
                 textcoords="offset points",
                 xytext=(0,10),
                 ha='center')

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve Across Confidence Thresholds')
plt.grid(True)
plt.tight_layout()
plt.savefig(os.path.join(eval_dir, "precision_recall_curve.png"))
plt.show()

# Plot metrics vs thresholds
plt.figure(figsize=(12, 8))
plt.plot(threshold_df["threshold"], threshold_df["precision"], 'o-', label='Precisi
plt.plot(threshold_df["threshold"], threshold_df["recall"], 'o-', label='Recall')
plt.plot(threshold_df["threshold"], threshold_df["mAP50"], 'o-', label='mAP50')
plt.plot(threshold_df["threshold"], threshold_df["f1"], 'o-', label='F1 Score')

plt.xlabel('Confidence Threshold')
plt.ylabel('Metric Value')
plt.title('Metrics vs Confidence Threshold')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig(os.path.join(eval_dir, "threshold_metrics.png"))
plt.show()
```

4. **Find Optimal Thresholds**

```
python
```

```python
# Find optimal threshold for different use cases
max_f1_idx = threshold_df["f1"].idxmax()
max_f1_threshold = threshold_df.loc[max_f1_idx, "threshold"]

max_map_idx = threshold_df["mAP50"].idxmax()
max_map_threshold = threshold_df.loc[max_map_idx, "threshold"]

balanced_idx = (threshold_df["precision"] - threshold_df["recall"]).abs().idxmin()
balanced_threshold = threshold_df.loc[balanced_idx, "threshold"]

# Print optimal thresholds
print("\nOptimal Thresholds:")
print(f"Best F1 Score: {threshold_df.loc[max_f1_idx, 'f1']:.4f} at threshold {max_f
print(f"Best mAP50: {threshold_df.loc[max_map_idx, 'mAP50']:.4f} at threshold {max_
print(f"Most Balanced (Precision ≈ Recall): threshold {balanced_threshold:.2f}")
print(f"  Precision: {threshold_df.loc[balanced_idx, 'precision']:.4f}")
print(f"  Recall: {threshold_df.loc[balanced_idx, 'recall']:.4f}")

# Save threshold analysis
threshold_analysis = {
    "thresholds": threshold_results,
    "optimal": {
        "max_f1": {
            "threshold": float(max_f1_threshold),
            "f1": float(threshold_df.loc[max_f1_idx, "f1"]),
            "precision": float(threshold_df.loc[max_f1_idx, "precision"]),
            "recall": float(threshold_df.loc[max_f1_idx, "recall"])
        },
        "max_map50": {
            "threshold": float(max_map_threshold),
            "mAP50": float(threshold_df.loc[max_map_idx, "mAP50"]),
            "precision": float(threshold_df.loc[max_map_idx, "precision"]),
            "recall": float(threshold_df.loc[max_map_idx, "recall"])
        },
        "balanced": {
            "threshold": float(balanced_threshold),
            "precision": float(threshold_df.loc[balanced_idx, "precision"]),
            "recall": float(threshold_df.loc[balanced_idx, "recall"])
        }
    }
}

# Save to file
threshold_file = os.path.join(eval_dir, "threshold_analysis.json")
with open(threshold_file, 'w') as f:
    json.dump(threshold_analysis, f, indent=2)
```

```
# Also save for model directory (dashboard integration)
model_threshold_file = os.path.join(model_dir, "threshold_analysis.json")
with open(model_threshold_file, 'w') as f:
    json.dump(threshold_analysis, f, indent=2)

print(f"\nThreshold analysis saved to {threshold_file}")
print(f"Also saved to model directory for dashboard integration")
```

03_evaluation/error_analysis.ipynb

Purpose: Analyze error patterns and generate confusion matrix

Key Sections:

1. **Load Model and Run Predictions**

```python
import os
import json
import yaml
import numpy as np
from ultralytics import YOLO
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from datetime import datetime

# Find latest model
models_dir = "/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/models/trained"
model_dirs = [d for d in os.listdir(models_dir) if os.path.isdir(os.path.join(model

# Use most recent model
latest_model = sorted(model_dirs, reverse=True)[0]
model_dir = os.path.join(models_dir, latest_model)
model_path = os.path.join(model_dir, "weights/best.pt")

# Load the model
model = YOLO(model_path)

# Find dataset
with open(os.path.join(model_dir, "args.yaml"), 'r') as f:
    training_args = yaml.safe_load(f)

data_yaml = training_args.get("data")

with open(data_yaml, 'r') as f:
    data_config = yaml.safe_load(f)

# Get validation images
val_images_dir = os.path.join(os.path.dirname(data_yaml), data_config["val"])
val_labels_dir = val_images_dir.replace("images", "labels")

# Create output directory
timestamp = datetime.now().strftime("%Y%m%d_%H%M")
error_dir = f"/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/reports/evaluation
os.makedirs(error_dir, exist_ok=True)
os.makedirs(os.path.join(error_dir, "examples"), exist_ok=True)
```

2. **Generate Confusion Matrix**

```
python
```

```python
# Run validation to get confusion matrix
val_results = model.val(data=data_yaml, conf=0.25, iou=0.7, task="val")

# Extract confusion matrix data
if hasattr(val_results, "confusion_matrix") and val_results.confusion_matrix is not
    conf_matrix = val_results.confusion_matrix.matrix
    class_names = list(val_results.names.values())

    # Create confusion matrix data
    matrix_data = {
        "matrix": conf_matrix.tolist(),
        "class_names": class_names
    }

    # Save confusion matrix
    matrix_file = os.path.join(error_dir, "confusion_matrix.json")
    with open(matrix_file, 'w') as f:
        json.dump(matrix_data, f, indent=2)

    # Also save to model directory for dashboard
    model_matrix_file = os.path.join(model_dir, "confusion_matrix.json")
    with open(model_matrix_file, 'w') as f:
        json.dump(matrix_data, f, indent=2)

    # Visualize confusion matrix
    plt.figure(figsize=(12, 10))

    # Limit to classes with some data
    non_zero_classes = []
    for i in range(len(class_names)):
        if conf_matrix[i].sum() > 0 or conf_matrix[:, i].sum() > 0:
            non_zero_classes.append(i)

    # Extract sub-matrix with non-zero classes
    sub_matrix = conf_matrix[non_zero_classes, :][:, non_zero_classes]
    sub_names = [class_names[i] for i in non_zero_classes]

    # Plot as heatmap
    plt.imshow(sub_matrix, cmap='Blues')

    # Add labels
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title('Confusion Matrix')

    plt.xticks(range(len(sub_names)), sub_names, rotation=90)
```

```
        plt.yticks(range(len(sub_names)), sub_names)

        # Add text annotations
        for i in range(len(sub_names)):
            for j in range(len(sub_names)):
                if sub_matrix[i, j] > 0:
                    plt.text(j, i, str(int(sub_matrix[i, j])),
                             ha="center", va="center",
                             color="white" if sub_matrix[i, j] > sub_matrix.max() / 2 e

        plt.tight_layout()
        plt.savefig(os.path.join(error_dir, "confusion_matrix.png"))

        # Analyze common misclassifications
        misclassifications = []
        for i in range(len(sub_names)):
            for j in range(len(sub_names)):
                if i != j and sub_matrix[i, j] > 0:
                    misclassifications.append({
                        "actual": sub_names[i],
                        "predicted": sub_names[j],
                        "count": int(sub_matrix[i, j])
                    })

        # Sort by count
        misclassifications.sort(key=lambda x: x["count"], reverse=True)

        # Display top misclassifications
        print("Top Misclassifications:")
        for i, m in enumerate(misclassifications[:10]):
            print(f"{i+1}. Actual: {m['actual']}, Predicted: {m['predicted']}, Count: {
    else:
        print("Confusion matrix not available from validation results")
```

3. **Identify and Visualize Error Examples**

```
python
```

```python
# Get validation image files
val_image_files = [f for f in os.listdir(val_images_dir) if f.lower().endswith(('.j

# Run predictions on validation set
results = model.predict(os.path.join(val_images_dir, "*.jpg"), conf=0.25, save=Fals

# Identify error cases
error_examples = []

for i, result in enumerate(results):
    image_path = result.path
    image_name = os.path.basename(image_path)

    # Get label path
    label_path = os.path.join(val_labels_dir, os.path.splitext(image_name)[0] + ".t

    # Load ground truth labels
    if os.path.exists(label_path):
        with open(label_path, 'r') as f:
            gt_lines = f.readlines()

        gt_labels = []
        for line in gt_lines:
            parts = line.strip().split()
            if len(parts) >= 5:
                class_id = int(parts[0])
                gt_labels.append({
                    "class_id": class_id,
                    "class_name": class_names[class_id] if class_id < len(class_nam
                    "x": float(parts[1]),
                    "y": float(parts[2]),
                    "w": float(parts[3]),
                    "h": float(parts[4])
                })

        # Get predictions
        pred_labels = []
        for box in result.boxes:
            class_id = int(box.cls.item())
            pred_labels.append({
                "class_id": class_id,
                "class_name": class_names[class_id] if class_id < len(class_names)
                "x": box.xywhn[0][0].item(),
                "y": box.xywhn[0][1].item(),
                "w": box.xywhn[0][2].item(),
                "h": box.xywhn[0][3].item(),
```

```
                "conf": box.conf.item()
            })

        # Check for misclassifications
        if len(gt_labels) != len(pred_labels):
            error_examples.append({
                "image": image_path,
                "type": "count_mismatch",
                "gt_count": len(gt_labels),
                "pred_count": len(pred_labels),
                "gt_labels": gt_labels,
                "pred_labels": pred_labels
            })
        else:
            # Match predictions to ground truth
            for gt in gt_labels:
                matched = False
                for pred in pred_labels:
                    # Check if same object (IOU > 0.5)
                    # Simplified check - just check if centers are close
                    dist = np.sqrt((gt["x"] - pred["x"])**2 + (gt["y"] - pred["y"])
                    if dist < 0.2:  # Simple threshold for center distance
                        if gt["class_id"] != pred["class_id"]:
                            error_examples.append({
                                "image": image_path,
                                "type": "misclassification",
                                "gt_label": gt,
                                "pred_label": pred
                            })
                        matched = True
                        break

                if not matched:
                    error_examples.append({
                        "image": image_path,
                        "type": "missed_detection",
                        "gt_label": gt
                    })

# Save error examples
error_file = os.path.join(error_dir, "error_examples.json")
with open(error_file, 'w') as f:
    json.dump(error_examples, f, indent=2)

print(f"\nFound {len(error_examples)} error examples")
print(f"Error analysis saved to {error_dir}")
```

4. **Generate Error Visualizations**

```
python
```

```python
# Visualize a few error examples
if error_examples:
    # Get misclassification examples
    misclass_examples = [e for e in error_examples if e["type"] == "misclassificati

    if misclass_examples:
        for i, example in enumerate(misclass_examples[:5]):  # Show top 5
            image_path = example["image"]
            img = Image.open(image_path)

            # Create figure
            fig, ax = plt.subplots(1, figsize=(10, 8))
            ax.imshow(img)

            # Get ground truth and prediction
            gt = example["gt_label"]
            pred = example["pred_label"]

            # Convert normalized coordinates to pixel coordinates
            img_w, img_h = img.size
            gt_x, gt_y = gt["x"] * img_w, gt["y"] * img_h
            gt_w, gt_h = gt["w"] * img_w, gt["h"] * img_h
            gt_left, gt_top = gt_x - gt_w/2, gt_y - gt_h/2

            pred_x, pred_y = pred["x"] * img_w, pred["y"] * img_h
            pred_w, pred_h = pred["w"] * img_w, pred["h"] * img_h
            pred_left, pred_top = pred_x - pred_w/2, pred_y - pred_h/2

            # Draw ground truth box (green)
            gt_rect = patches.Rectangle((gt_left, gt_top), gt_w, gt_h,
                                        linewidth=2, edgecolor='green', facecolor='n
            ax.add_patch(gt_rect)
            ax.text(gt_left, gt_top-10, f"GT: {gt['class_name']}", color='green',
                    backgroundcolor='white', fontsize=12)

            # Draw prediction box (red)
            pred_rect = patches.Rectangle((pred_left, pred_top), pred_w, pred_h,
                                          linewidth=2, edgecolor='red', facecolor='no
            ax.add_patch(pred_rect)
            ax.text(pred_left, pred_top+pred_h+10,
                    f"Pred: {pred['class_name']} ({pred['conf']:.2f})",
                    color='red', backgroundcolor='white', fontsize=12)

            # Set title
            ax.set_title(f"Misclassification Example {i+1}: {gt['class_name']} → {p
```

```
            # Remove axis
            ax.axis('off')

            # Save figure
            plt.tight_layout()
            error_img_path = os.path.join(error_dir, "examples", f"misclass_{i+1}.p
            plt.savefig(error_img_path)
            plt.close()

        print(f"Generated {min(5, len(misclass_examples))} misclassification visual
```

04_dashboard_integration/metrics_generation.ipynb

Purpose: Generate and validate dashboard-compatible metrics files

Key Sections:

1. **Find and Load Latest Evaluation Results**

```python
import os
import json
import glob
import pandas as pd
from datetime import datetime

# Find latest evaluation directory
reports_dir = "/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/reports"
eval_dirs = glob.glob(os.path.join(reports_dir, "evaluation_*"))

if not eval_dirs:
    print("Error: No evaluation directories found")
else:
    # Use most recent evaluation
    latest_eval = sorted(eval_dirs, reverse=True)[0]
    print(f"Using evaluation results from: {latest_eval}")

    # Find latest model directory
    models_dir = "/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/models/trained
    model_dirs = [d for d in os.listdir(models_dir) if os.path.isdir(os.path.join(m

    if not model_dirs:
        print("Error: No model directories found")
    else:
        # Use most recent model
        latest_model = sorted(model_dirs, reverse=True)[0]
        model_dir = os.path.join(models_dir, latest_model)
        print(f"Using model directory: {model_dir}")
```

2. **Load and Format Evaluation Metrics**

```
python
```

```python
# Load performance metrics
metrics_file = os.path.join(latest_eval, "performance_metrics.json")

if os.path.exists(metrics_file):
    with open(metrics_file, 'r') as f:
        metrics = json.load(f)
    print("Loaded performance metrics")
else:
    print(f"Error: Metrics file not found at {metrics_file}")
    metrics = {}

# Load confusion matrix
matrix_file = os.path.join(latest_eval, "confusion_matrix.json")

if os.path.exists(matrix_file):
    with open(matrix_file, 'r') as f:
        confusion_matrix = json.load(f)
    print("Loaded confusion matrix")
else:
    print(f"Error: Confusion matrix file not found at {matrix_file}")
    confusion_matrix = {"matrix": [], "class_names": []}

# Load threshold analysis
threshold_dirs = glob.glob(os.path.join(reports_dir, "evaluation_*_thresholds"))

if threshold_dirs:
    # Use most recent threshold analysis
    latest_threshold = sorted(threshold_dirs, reverse=True)[0]
    threshold_file = os.path.join(latest_threshold, "threshold_analysis.json")

    if os.path.exists(threshold_file):
        with open(threshold_file, 'r') as f:
            threshold_analysis = json.load(f)
        print("Loaded threshold analysis")

        # Add thresholds to metrics
        if "thresholds" in threshold_analysis:
            metrics["thresholds"] = threshold_analysis["thresholds"]
    else:
        print(f"Warning: Threshold file not found at {threshold_file}")
```

3. **Format Data for Dashboard**

```python
# Get model arguments
args_file = os.path.join(model_dir, "args.yaml")

if os.path.exists(args_file):
    import yaml
    with open(args_file, 'r') as f:
        model_args = yaml.safe_load(f)
else:
    model_args = {}

# Create model details
model_details = {
    "model_name": os.path.basename(model_dir),
    "model_type": "YOLOv8",
    "created_at": datetime.fromtimestamp(os.path.getctime(model_dir)).strftime("%Y-
    "image_size": model_args.get("imgsz", 640),
    "batch_size": model_args.get("batch", 16),
    "config": model_args
}

# Create training history if not already in metrics
if "history" not in metrics:
    # Try to read from results.csv
    results_csv = os.path.join(model_dir, "results.csv")

    if os.path.exists(results_csv):
        results_df = pd.read_csv(results_csv)

        # Find metric columns
        precision_col = next((col for col in results_df.columns if 'precision' in c
        recall_col = next((col for col in results_df.columns if 'recall' in col.low
        map50_col = next((col for col in results_df.columns if 'map50' in col.lower
        map50_95_col = next((col for col in results_df.columns if 'map50-95' in col

        # Create training history
        metrics["history"] = {
            "epoch": results_df["epoch"].tolist(),
            "precision": results_df[precision_col].tolist() if precision_col else [
            "recall": results_df[recall_col].tolist() if recall_col else [],
            "mAP50": results_df[map50_col].tolist() if map50_col else [],
            "mAP50-95": results_df[map50_95_col].tolist() if map50_95_col else []
        }
```

4. **Generate Dashboard Files**

```python
# Create dashboard files
dashboard_files = {
    "performance_metrics.json": metrics,
    "class_metrics.json": metrics.get("per_class", {}),
    "confusion_matrix.json": confusion_matrix,
    "training_history.json": metrics.get("history", {"epoch": []}),
    "model_details.json": model_details
}

# Create timestamp for backup
timestamp = datetime.now().strftime("%Y%m%d_%H%M")

# Save files to model directory
for filename, data in dashboard_files.items():
    # Create backup of existing file if it exists
    target_file = os.path.join(model_dir, filename)
    if os.path.exists(target_file):
        backup_file = f"{target_file}.{timestamp}.bak"
        os.rename(target_file, backup_file)
        print(f"Created backup of {filename}: {backup_file}")

    # Save new file
    with open(target_file, 'w') as f:
        json.dump(data, f, indent=2)
    print(f"Saved {filename} to model directory")

print("\nDashboard files generated successfully!")
```

04_dashboard_integration/dashboard_preview.ipynb

Purpose: Preview dashboard with generated metrics

Key Sections:

1. **Load Dashboard Files**

```python
import os
import json
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Find latest model directory
models_dir = "/home/peter/Desktop/TU PHD/WildlifeDetectionSystem/models/trained"
model_dirs = [d for d in os.listdir(models_dir) if os.path.isdir(os.path.join(model

if not model_dirs:
    print("Error: No model directories found")
else:
    # Use most recent model
    latest_model = sorted(model_dirs, reverse=True)[0]
    model_dir = os.path.join(models_dir, latest_model)
    print(f"Using model directory: {model_dir}")

    # Load dashboard files
    dashboard_files = {
        "performance_metrics": os.path.join(model_dir, "performance_metrics.json"),
        "class_metrics": os.path.join(model_dir, "class_metrics.json"),
        "confusion_matrix": os.path.join(model_dir, "confusion_matrix.json"),
        "training_history": os.path.join(model_dir, "training_history.json"),
        "model_details": os.path.join(model_dir, "model_details.json")
    }

    # Check if files exist
    missing_files = []
    for name, path in dashboard_files.items():
        if not os.path.exists(path):
            print(f"Missing dashboard file: {name}")
            missing_files.append(name)

    if missing_files:
        print(f"Warning: {len(missing_files)} dashboard files are missing")
    else:
        print("All dashboard files are present")
```

2. **Validate Dashboard Files**

```
python
```

```
# Load and validate dashboard files
dashboard_data = {}

for name, path in dashboard_files.items():
    if os.path.exists(path):
        try:
            with open(path, 'r') as f:
                data = json.load(f)
                dashboard_data[name] = data
            print(f"✅ {name}: Valid JSON format")
        except json.JSONDecodeError as e:
            print(f"❌ {name}: Invalid JSON format - {e}")

# Check required fields in performance_metrics
if "performance_metrics" in dashboard_data:
    metrics = dashboard_data["performance_metrics"]
    required_fields = ["precision", "recall", "mAP50", "mAP50-95"]
    missing_fields = [field for field in required_fields if field not in metrics]

    if missing_fields:
        print(f"❌ performance_metrics: Missing required fields: {', '.join(missing
    else:
        print("✅ performance_metrics: All required fields present")

        # Check if values are numbers
        invalid_fields = []
        for field in required_fields:
            if not isinstance(metrics[field], (int, float)):
                invalid_fields.append(field)

        if invalid_fields:
            print(f"❌ performance_metrics: Non-numeric values in fields: {', '.joi
        else:
            print("✅ performance_metrics: All values are numeric")

# Check confusion_matrix format
if "confusion_matrix" in dashboard_data:
    conf_matrix = dashboard_data["confusion_matrix"]

    if "matrix" not in conf_matrix or "class_names" not in conf_matrix:
        print("❌ confusion_matrix: Missing 'matrix' or 'class_names' fields")
    else:
        print("✅ confusion_matrix: Required fields present")

        # Check if matrix and class_names have matching dimensions
        matrix = conf_matrix["matrix"]
```

```
        class_names = conf_matrix["class_names"]

        if not matrix:
            print("❌ confusion_matrix: Empty matrix")
        elif len(matrix) != len(class_names):
            print(f"❌ confusion_matrix: Dimension mismatch - matrix rows: {len(mat
        else:
            print("✅ confusion_matrix: Dimensions match")

# Check training_history format
if "training_history" in dashboard_data:
    history = dashboard_data["training_history"]

    if "epoch" not in history:
        print("❌ training_history: Missing 'epoch' field")
    else:
        print("✅ training_history: 'epoch' field present")

        # Check if all arrays have same length
        epoch_len = len(history["epoch"])
        different_length = []

        for key, value in history.items():
            if key != "epoch" and len(value) != epoch_len:
                different_length.append(f"{key}: {len(value)}")

        if different_length:
            print(f"❌ training_history: Length mismatch - epoch: {epoch_len}, {',
        else:
            print("✅ training_history: All arrays have same length")
```

3. **Preview Performance Metrics**

```
python
```

```python
# Create performance preview
if "performance_metrics" in dashboard_data:
    metrics = dashboard_data["performance_metrics"]

    # Display overall metrics
    print("\n=== Overall Performance ===")
    print(f"Precision: {metrics['precision']*100:.1f}%")
    print(f"Recall: {metrics['recall']*100:.1f}%")
    print(f"mAP@0.5: {metrics['mAP50']*100:.1f}%")
    print(f"mAP@0.5:0.95: {metrics['mAP50-95']*100:.1f}%")

    # Calculate F1 score
    if metrics['precision'] > 0 and metrics['recall'] > 0:
        f1 = 2 * metrics['precision'] * metrics['recall'] / (metrics['precision'] +
        print(f"F1 Score: {f1*100:.1f}%")

    # Display training info
    if "training_epochs" in metrics and "best_epoch" in metrics:
        print(f"\nTraining Epochs: {metrics['training_epochs']}")
        print(f"Best Epoch: {metrics['best_epoch']}")

    # Get top and bottom classes
    if "per_class" in metrics and metrics["per_class"]:
        per_class = metrics["per_class"]

        # Create DataFrame for better analysis
        class_data = []
        for class_name, class_metrics in per_class.items():
            class_data.append({
                "Class": class_name,
                "Precision": class_metrics.get("precision", 0),
                "Recall": class_metrics.get("recall", 0),
                "mAP50": class_metrics.get("map50", 0)
            })

        class_df = pd.DataFrame(class_data)

        # Calculate F1 score
        class_df["F1"] = 2 * class_df["Precision"] * class_df["Recall"] / (class_df
        class_df["F1"] = class_df["F1"].fillna(0)  # Handle division by zero

        # Top 5 classes by mAP50
        print("\n=== Top 5 Classes by mAP50 ===")
        top_classes = class_df.sort_values("mAP50", ascending=False).head(5)
        for _, row in top_classes.iterrows():
            print(f"{row['Class']}: mAP50={row['mAP50']*100:.1f}%, Precision={row['
```

```python
# Bottom 5 classes by mAP50
print("\n=== Bottom 5 Classes by mAP50 ===")
bottom_classes = class_df.sort_values("mAP50").head(5)
for _, row in bottom_classes.iterrows():
    print(f"{row['Class']}: mAP50={row['mAP50']*100:.1f}%, Precision={row['

# Visualize class performance
plt.figure(figsize=(12, 8))
top10_classes = class_df.sort_values("mAP50", ascending=False).head(10)

# Create bar chart
x = range(len(top10_classes))
width = 0.25

plt.bar([i - width for i in x], top10_classes["Precision"] * 100, width, la
plt.bar(x, top10_classes["Recall"] * 100, width, label='Recall')
plt.bar([i + width for i in x], top10_classes["mAP50"] * 100, width, label=

plt.xlabel('Class')
plt.ylabel('Percentage (%)')
plt.title('Top 10 Classes Performance')
plt.xticks(x, top10_classes["Class"], rotation=45, ha='right')
plt.legend()
plt.tight_layout()
plt.show()
```

4. **Preview Confusion Matrix**

```
python
```

```python
# Preview confusion matrix
if "confusion_matrix" in dashboard_data:
    conf_matrix = dashboard_data["confusion_matrix"]

    if conf_matrix["matrix"] and conf_matrix["class_names"]:
        matrix = np.array(conf_matrix["matrix"])
        class_names = conf_matrix["class_names"]

        # Find non-empty classes (rows or columns with non-zero values)
        non_empty = []
        for i in range(len(class_names)):
            if matrix[i].sum() > 0 or matrix[:, i].sum() > 0:
                non_empty.append(i)

        if non_empty:
            # Extract sub-matrix with non-empty classes
            sub_matrix = matrix[non_empty, :][:, non_empty]
            sub_names = [class_names[i] for i in non_empty]

            # Limit to at most 15 classes for better visualization
            if len(sub_names) > 15:
                # Use top 15 by total count
                class_totals = sub_matrix.sum(axis=1) + sub_matrix.sum(axis=0)
                top_indices = np.argsort(class_totals)[-15:]
                sub_matrix = sub_matrix[top_indices, :][:, top_indices]
                sub_names = [sub_names[i] for i in top_indices]

            # Normalize matrix for better visualization
            row_sums = sub_matrix.sum(axis=1)
            normalized_matrix = np.zeros_like(sub_matrix, dtype=float)
            for i in range(len(sub_matrix)):
                if row_sums[i] > 0:
                    normalized_matrix[i] = sub_matrix[i] / row_sums[i]

            # Create heatmap
            plt.figure(figsize=(12, 10))
            sns.heatmap(normalized_matrix, annot=sub_matrix.astype(int),
                        fmt='d', cmap='Blues', xticklabels=sub_names, yticklabels=su
            plt.xlabel('Predicted')
            plt.ylabel('Actual')
            plt.title('Confusion Matrix (normalized by row)')
            plt.xticks(rotation=45, ha='right')
            plt.tight_layout()
            plt.show()

            # Find top misclassifications
```

```
misclassifications = []
for i in range(len(sub_names)):
    for j in range(len(sub_names)):
        if i != j and sub_matrix[i, j] > 0:
            misclassifications.append({
                "actual": sub_names[i],
                "predicted": sub_names[j],
                "count": int(sub_matrix[i, j])
            })

# Sort by count
misclassifications.sort(key=lambda x: x["count"], reverse=True)

# Display top misclassifications
print("\n=== Top Misclassifications ===")
for i, m in enumerate(misclassifications[:5]):
    print(f"{i+1}. Actual: {m['actual']}, Predicted: {m['predicted']},
```

Dashboard File Structure

The files needed for your dashboard:

1. **performance_metrics.json**

json


```
{
  "precision": 0.637,
  "recall": 0.409,
  "mAP50": 0.505,
  "mAP50-95": 0.313,
  "training_epochs": 60,
  "best_epoch": 35,
  "classes": 30,
  "per_class": {
    "Male Roe Deer": {"precision": 0.823, "recall": 0.404, "map50": 0.713},
    "Female Roe Deer": {"precision": 0.301, "recall": 0.786, "map50": 0.322},
    "Fox": {"precision": 0.425, "recall": 0.375, "map50": 0.291}
    // Other classes...
  },
  "thresholds": [
    {"threshold": 0.05, "precision": 0.454, "recall": 0.414, "mAP50": 0.357},
    {"threshold": 0.1, "precision": 0.328, "recall": 0.414, "mAP50": 0.356},
    {"threshold": 0.25, "precision": 0.328, "recall": 0.414, "mAP50": 0.355},
    {"threshold": 0.5, "precision": 0.336, "recall": 0.413, "mAP50": 0.353},
    {"threshold": 0.75, "precision": 0.637, "recall": 0.299, "mAP50": 0.317},
    {"threshold": 0.9, "precision": 0.822, "recall": 0.187, "mAP50": 0.232}
  ],
  "history": {
    "epoch": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60
    "precision": [0.01, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.52
    "recall": [0.01, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.32, 0.35, 0.37, 0.39, 0.4,
    "mAP50": [0.01, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.32, 0.35, 0.37, 0.4, 0.43, 0
    "mAP50-95": [0.005, 0.02, 0.05, 0.08, 0.1, 0.15, 0.18, 0.2, 0.22, 0.24, 0.26, 0
  }
}
```


2. **class_metrics.json**

json


```
{
  "Male Roe Deer": {"precision": 0.823, "recall": 0.404, "map50": 0.713},
  "Female Roe Deer": {"precision": 0.301, "recall": 0.786, "map50": 0.322},
  "Fox": {"precision": 0.425, "recall": 0.375, "map50": 0.291},
  "Jackal": {"precision": 0.329, "recall": 0.25, "map50": 0.321},
  "Weasel": {"precision": 1.0, "recall": 0.0, "map50": 0.995},
  "Wildcat": {"precision": 1.0, "recall": 0.0, "map50": 0.0},
  "Rabbit": {"precision": 0.511, "recall": 0.741, "map50": 0.601},
  "Human": {"precision": 0.704, "recall": 0.716, "map50": 0.797}
}
```

3. **confusion_matrix.json**

```json
{
  "matrix": [
    [10, 2, 0, 0, 0, 0, 0, 0],
    [1, 15, 3, 0, 0, 0, 0, 0],
    [0, 4, 20, 2, 0, 0, 0, 0],
    [0, 0, 3, 12, 1, 0, 0, 0],
    [0, 0, 0, 0, 8, 2, 0, 0],
    [0, 0, 0, 0, 1, 7, 0, 0],
    [0, 0, 0, 0, 0, 0, 22, 1],
    [0, 0, 0, 0, 0, 0, 2, 18]
  ],
  "class_names": [
    "Male Roe Deer",
    "Female Roe Deer",
    "Fox",
    "Jackal",
    "Weasel",
    "Wildcat",
    "Rabbit",
    "Human"
  ]
}
```

4. **training_history.json**

```json
{
  "epoch": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60],
  "precision": [0.01, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.52,
  "recall": [0.01, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.32, 0.35, 0.37, 0.39, 0.4, 0.
  "mAP50": [0.01, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.32, 0.35, 0.37, 0.4, 0.43, 0.4
  "mAP50-95": [0.005, 0.02, 0.05, 0.08, 0.1, 0.15,
```