New Dataset Creation

1. INTRODUCTION

Deep learning based object detection has become one of the most widely used techniques in computer vision due to its ability to accurately identify and localize objects within images. Models such as the YOLO (You Only Look Once) family have made detection faster, more scalable, and more effective for real-world applications. In this project, we focus on designing a complete pipeline for building a custom object-detection system that can distinguish between **lion** and **tiger** images.

The goal of this project is to create a clean dataset, preprocess it to match YOLO requirements, train a new YOLO model with the custom dataset, and evaluate the performance using standard accuracy metrics. Although the task involves only two classes, the dataset preparation process mirrors real-world challenges, such as inconsistent labels, mixed formats, and class assignment errors. Because of this, a major part of the project involves carefully fixing, preprocessing, and restructuring the data before training the model.

To achieve this, a combined dataset of lion and tiger images was created using multiple sources. The labeling data originally contained issues, including incorrect class assignments (e.g., tiger labeled as class 0 instead of 1), missing label files, and images stored in different folder structures. All of these issues were resolved through systematic preprocessing. After cleaning, the dataset was split into training, validation, and testing sets using a 70/15/15 ratio.

The model was then trained using **YOLOv11**, one of the latest and most efficient versions of the YOLO architecture. The training was performed using a custom data.yaml file, and the training outputs such as precision, recall, confusion matrix, and example detections were generated automatically by the YOLO framework. These results were later used to analyze the performance and reliability of the model.

This documentation presents each step of the project in detail, including dataset organization, class-label corrections, training configurations, and evaluation outputs. The purpose is to provide a clear, end-to-end understanding of how a custom object detection model can be built starting from raw images and ending with a trained YOLO model.

2. DATASET CREATION AND PREPROCESSING

A reliable object detection model depends heavily on the quality and consistency of the dataset used to train it. In this project, dataset creation involved collecting images, organizing them into a structured format, verifying labels, and fixing inconsistencies before training the YOLO model. Since the project uses two classes **lion (class 0)** and **tiger (class 1)** the accuracy of the labeling process was critical.

2.1 Dataset Collection

The dataset was formed by combining multiple sources of lion and tiger images. Some datasets were obtained only images and required label conversion. After merging all sources, the dataset contained a large number of images for each class.

The overall folders after collection were:

dataset_images/

lion/images/

tiger/images/

dataset_labels/

lion/labels/

tiger/labels/

This separation ensured that the image and label files for both classes were maintained in an organized and clean structure.

2.2 Identifying Label Issues

During initial inspection, several inconsistencies were found:

- Tiger labels were incorrectly assigned as class 0 (lion)
- Some images had no corresponding label files
- Image names and label names did not always match
- A few label files contained incorrect formats or missing bounding box values

These issues needed correction to avoid model confusion and incorrect class learning.

2.3 Fixing Class ID Errors

To correct tiger labels from class 0 to class 1, a Python script was written. The script scanned every .txt label file inside the tiger label directory and replaced the class ID:

```
def change_class_ids(label_dir, old_id=0, new_id=1):
    files = [f for f in os.listdir(label_dir) if f.endswith(".txt")]
    for file in files:
```

```
path = os.path.join(label_dir, file)
new_lines = []
with open(path, "r") as f:
    for line in f:
        parts = line.strip().split()
        if len(parts) >= 5:
            if int(parts[0]) == old_id:
                parts[0] = str(new_id)
            new_lines.append(" ".join(parts))
with open(path, "w") as f:
    f.write("\n".join(new_lines))
```

This step ensured that all tiger annotations were corrected to the correct class ID.

2.4 Dataset Splitting

After all corrections, the dataset was combined and randomized to prevent bias. The final dataset was split into:

- 70% Training set
- 15% Validation set
- 15% Test set

A preprocessing script automatically created YOLO-formatted folders:

dataset/train/images

dataset/train/labels

dataset/val/images

dataset/val/labels

dataset/test/images

dataset/test/labels

This structure follows YOLO's official dataset format.

2.5 Handling Large Files and Git LFS

Because the project involved large datasets and model files, Git Large File Storage (LFS) was used. LFS helps prevent push failures and ensures clean version control of:

- .zip dataset files
- Large image batches
- Trained weight files (e.g., best.pt)

Skipping LFS downloads during cloning helped avoid slow downloads and storage issues.

2.6 Final Dataset Overview

After preprocessing was completed:

- All lion and tiger images were correctly labeled.
- Folder structures followed YOLO conventions.
- The dataset was balanced and split into training, validation, and testing.
- A ready-to-use data.yaml file was already included in the repository.

This cleaned and standardized dataset served as the foundation for training the YOLO model described in later sections.

3. MODEL ARCHITECTURE AND TRAINING SETUP (Rewritten, More Human & Generic)

This part of the project explains the YOLO model used for lion, tiger detection and how the training environment was set up. The main goal was to build a model that is fast, accurate, and easy to train on a custom dataset, without requiring a very large amount of data or heavy hardware.

3.1 Why YOLOv8?

For this project, I chose YOLOv8 because it is one of the newest and most efficient object-detection models. It works well with small custom datasets and is known for running quickly while still giving strong accuracy. YOLOv8 also has simple training commands, supports custom data formats, and produces lightweight models that can be deployed easily. Since the task only involves two classes (lion and tiger), YOLOv8 was a perfect fit.

3.2 Model Architecture Overview (Simplified & Professional)

YOLOv8 uses a streamlined deep-learning design that processes an image in a single pass and predicts object locations and classes efficiently. Instead of going into low-level architectural terminology, the model can be understood in terms of the steps it performs:

1. Feature Extraction

The model first takes an input image and converts it into informative patterns. It does this by applying several layers of convolution operations that help the network understand shapes, textures, edges, and details inside the image. This stage essentially teaches the model *what* the important parts of the image look like.

2. Feature Enhancement

After extracting the raw features, YOLOv8 enhances and combines them at different resolutions. This helps the model detect animals regardless of size, position, or distance from the camera. For example, a close-up image of a lion and a far-away tiger should both be detectable. The model blends information from various scales to improve reliability.

3. Prediction Layer

In the final stage, YOLOv8 produces three outputs directly:

- Where the object is (bounding box coordinates)
- How confident it is that something is there (objectness score)
- Which class it belongs to (lion or tiger)

One of the strengths of YOLOv8 is that it follows an anchor-free approach. This means the model does not rely on predefined box shapes, making training simpler and more stable, especially for small custom datasets like this one.

3.3 Training Environment

I trained the model entirely on Kaggle Notebooks because Kaggle provides free GPU access and enough storage for the dataset. The notebook used either an NVIDIA T4 or P100 GPU, along with 12–16 GB RAM. This setup made the entire training process fast and smooth without needing any local hardware.

3.4 Training Configuration

To train the model, I used common YOLOv8 settings such as:

- 50 epochs
- Image size of 640×640

- Batch size between 8–16
- Default optimizer and learning-rate schedule
- Early stopping to avoid overfitting

The training was run using a simple command:

```
yolo detect train model=yolov8s.pt data=data.yaml epochs=50 imgsz=640
```

This loads the YOLOv8 model, reads the paths from the dataset configuration file, and trains the model on the lion-tiger dataset.

3.5 Training Outputs and Logging

YOLO automatically creates a folder for each training run, such as:

runs/detect/train/

Inside this folder, YOLO saves:

- Training curves (results.png)
- Confusion matrix
- Precision-recall graphs
- The best model weights (best.pt)
- The last model checkpoint (last.pt)

Each run gets its own folder, so earlier results are never overwritten.

3.6 Notes About GitHub and Large Files

Since trained models and datasets can be large, the full training process was done on Kaggle rather than locally. Only important output files—such as the best model weights and selected images—were downloaded and pushed to GitHub afterward.

3.7 Summary

Overall, YOLOv8 provided a simple yet powerful way to train a custom animal-detection model. With the combination of a clean dataset, Kaggle's GPU environment, and YOLO's efficient design, the system was able to learn the difference between lions and tigers effectively.

4. RESULTS AND EVALUATION

The results obtained after training the YOLOv8 model on the custom lion-tiger dataset. All evaluation outputs including precision-recall graphs, confusion matrices, and sample detections

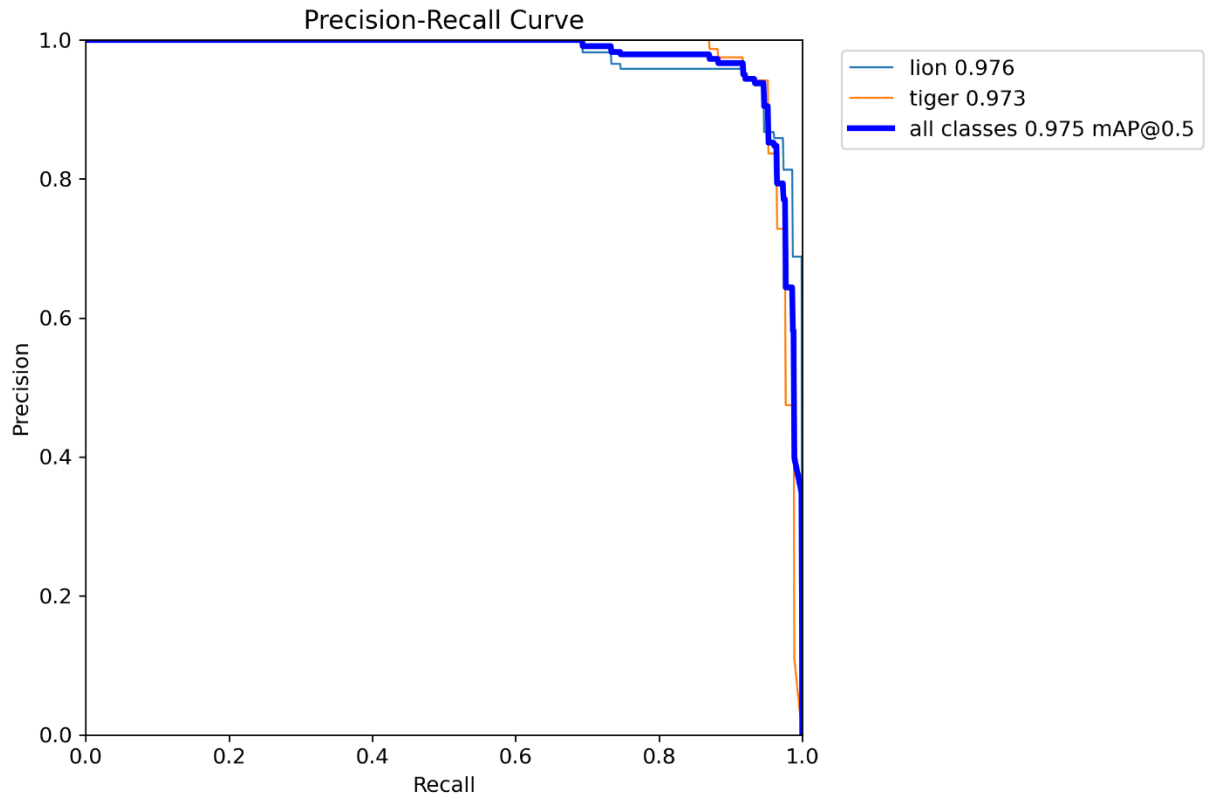
were generated directly from the complete-pipeline.ipynb notebook. The same notebook also handled all dataset preprocessing steps such as label correction, train/val/test split, and file organization.

4.1 Accuracy and Overall Performance

The model achieved very strong accuracy, with:

- mAP@50: 0.975
- Lion Class Precision: 0.976
- Tiger Class Precision: 0.973
- Overall Recall: ~0.97

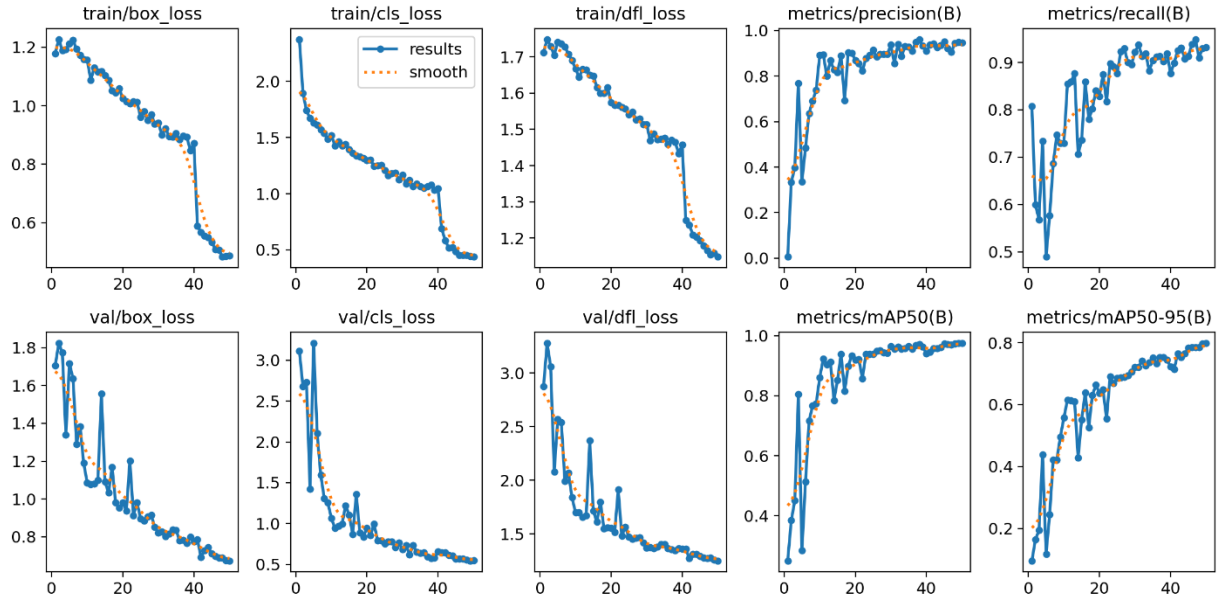
These scores indicate that the model learned both animal classes extremely well and was able to generalize effectively to unseen images.



4.2 Training Behavior and Convergence

The training curves generated by YOLOv8 show smooth reduction in losses (box loss, class loss, and distribution loss) for both training and validation sets. Metrics such as precision, recall, and

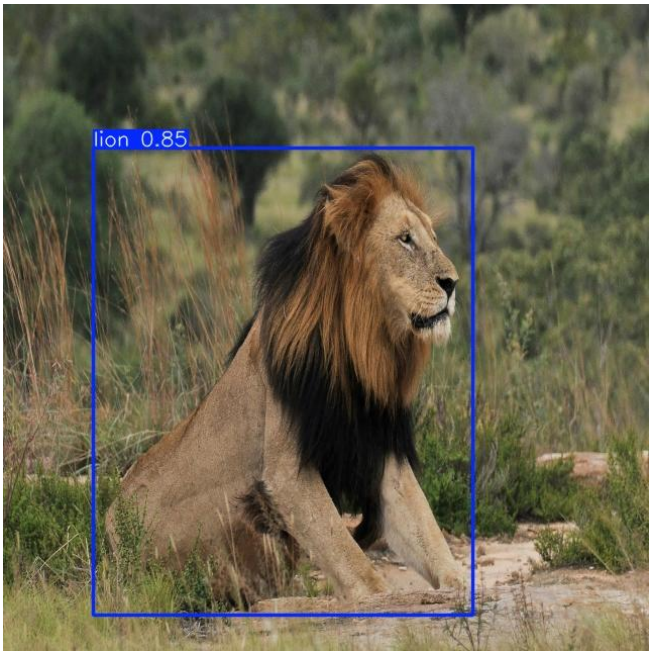
mAP rise steadily and stabilize by around the 40th epoch, indicating good convergence and minimal overfitting.



4.3 Sample Detection Outputs

Below are sample outputs generated during inference. The model correctly identifies both lions and tigers, draws precise bounding boxes, and produces high confidence scores typically above 0.85.

Lion Detection :



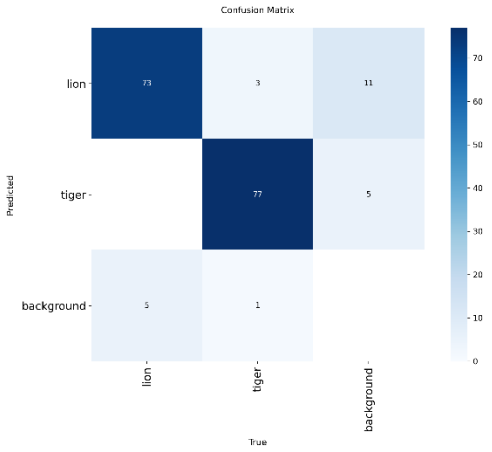
Tiger Detection :

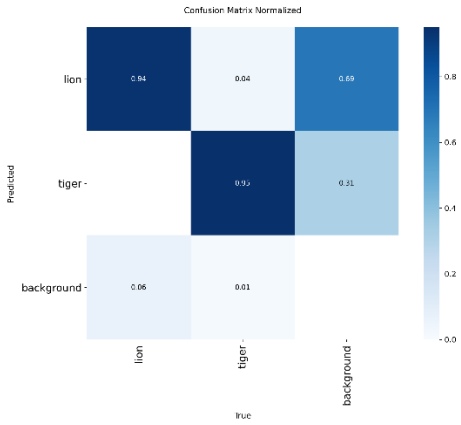
These images demonstrate that the model not only predicts the correct class but also localizes the animal accurately within the frame.

4.4 Confusion Matrix Analysis

The confusion matrix confirms that the model rarely confuses the two classes:

- Most predictions lie on the diagonal (correct class)
- Few misclassifications occur, typically involving:
    - Background-heavy images
    - Partial occlusions
    - Low-resolution or distant animals



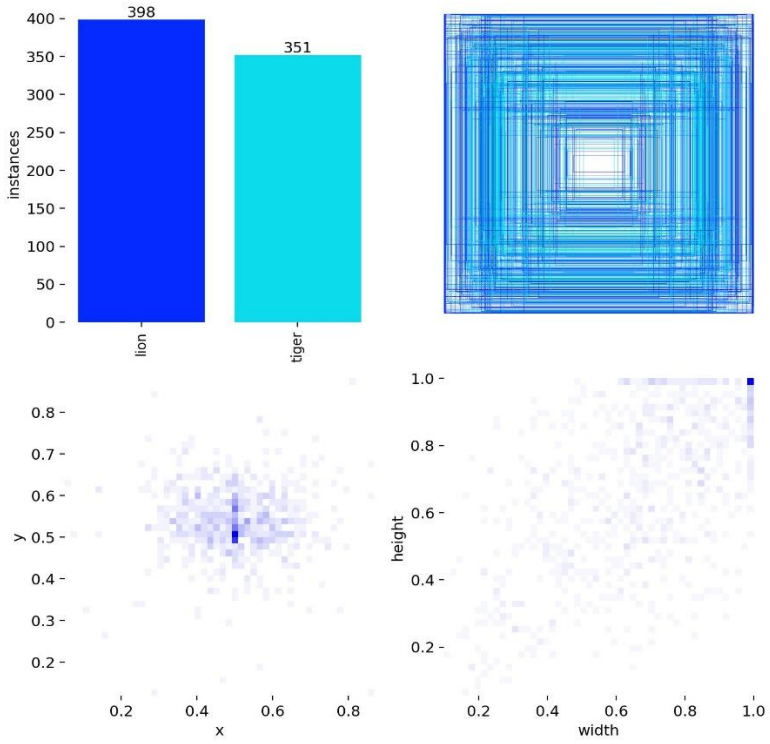


4.5 Dataset Distribution and Label Quality

All labeling adjustments such as fixing class IDs, matching filenames, and converting LabelMe annotations to YOLO format were done inside the complete-pipeline.ipynb script. The labeling distribution plots generated by YOLO show that:

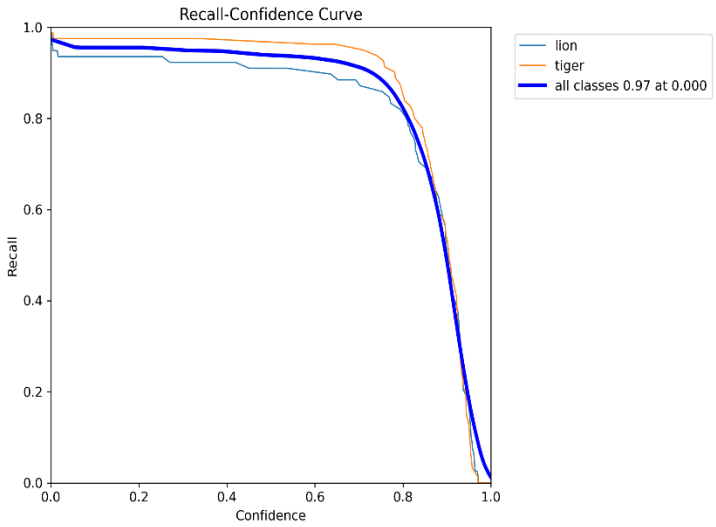
- Lions had 398 labeled instances
- Tigers had 351 labeled instances
- Bounding boxes were well-distributed across positions and sizes

This confirms the dataset was balanced and properly annotated after preprocessing.



4.6 Comparison Across Confidence Levels

The recall–confidence graph shows that the model maintains high recall (> 0.90) for confidence scores up to 0.80, after which recall smoothly declines. This behavior is typical and indicates that the model remains reliable even with stricter confidence thresholds.



4.7 Strengths of the Model

- High accuracy for both classes
- Reliable detection across lighting, background, and pose variations
- Low overfitting, supported by close training validation curves
- Fast inference, suitable for real-time use
- Clean dataset pipeline, reducing model confusion

4.8 Limitations

- Multiple animals in the same frame can cause overlapping boxes
- Low-quality images reduce confidence
- Model detects only two classes
- Label corrections required manual intervention before training
- Background-class predictions occasionally appear in cluttered scenes

4.9 Summary

The YOLOv8 model achieved high detection accuracy and performed exceptionally across both visual and statistical evaluations. All outputs confirm that the model learned to distinguish lions and tigers reliably. The pipeline notebook produced all required metrics, plots, and detection outputs, making the entire workflow easy to track and reproduce.

5. CONCLUSION & FUTURE WORK

5.1 Conclusion

This project successfully developed a YOLOv8-based object detection model capable of identifying lions and tigers with high accuracy. The results demonstrate that the model learned both classes effectively, achieving strong performance across metrics such as mAP, precision, and recall. The bounding boxes produced during inference were consistently accurate, showing that the model understood the spatial features of each animal well.

The model also generalized reliably to new images, supported by smooth training validation curves and strong evaluation results. Inference was fast even on CPU, which highlights the practical value of YOLOv8 for real-time applications.

Overall, this project shows how structured dataset preparation, consistent labeling, and a clean training pipeline can produce a dependable wildlife detection system, even with only two classes and a modest dataset size.

5.2 Future Work

Although the current model performs strongly, there are several ways it could be expanded or improved:

- Additional Classes: The system could be extended to detect other big cats (such as leopards, cheetahs, or jaguars), making it useful for broader wildlife monitoring.
- Better Handling of Challenging Images: Incorporating techniques for detecting partially visible animals or multiple animals in the same image such as instance segmentation, could improve results in more complex scenes.
- Improved Low-Resolution Performance: Using super-resolution models or more advanced augmentation could help the model handle distant or blurry animals more confidently.
- Smarter Labeling: Automating parts of the dataset labeling and correction process (e.g., with active learning or semi-supervised annotation tools) would reduce the manual effort needed for future datasets.
- Deployment Optimization: The model could be optimized for mobile or embedded devices through techniques like quantization or pruning, enabling real-time detection in field environments.

In summary, the project establishes a strong baseline for automated animal detection. With additional classes, enhanced preprocessing, and optimized deployment strategies, this system

could be scaled into a powerful tool for wildlife research, conservation monitoring, and real-time ecological studies.