

Start coding or [generate](#) with AI.

Task

Compare the object detection performance of two YOLOv8 models, "Trained_yolov8.pt" and "Kaggle_yolov8s.pt", on the following image files: "mango.png", "orange.jpg", "grapes.jpg", "banana.jpg", "strawberry.jpg", "watermelon.jpg", "apple.png", and "pineapple.jpg". The comparison should involve visualizing their predictions, extracting and comparing quantitative metrics such as the number of detections, types of objects detected, and confidence scores, generating comparison graphs, and summarizing the findings while acknowledging the limitations due to the absence of ground truth data.

Setup Environment and Define Resources

Subtask:

Install the `ultralytics` library, import necessary modules, and define the file paths for both YOLOv8 models (`Trained_yolov8.pt`, `Kaggle_yolov8s.pt`) and the list of image files (`mango.png`, `orange.jpg`, `grapes.jpg`, `banana.jpg`, `strawberry.jpg`, `watermelon.jpg`, `apple.png`, `pineapple.jpg`).

Reasoning: The first step is to install the `ultralytics` library as specified in the instructions. This requires a separate code block due to the use of `pip install`.

```
pip install ultralytics
```

```
Requirement already satisfied: ultralytics in /usr/local/lib/python3.12/dist-packages (8.3.253)
Requirement already satisfied: numpy>=1.23.0 in /usr/local/lib/python3.12/dist-packages (from ultralytics) (2.0.2)
Requirement already satisfied: matplotlib>=3.3.0 in /usr/local/lib/python3.12/dist-packages (from ultralytics) (3.3.0)
Requirement already satisfied: opencv-python>=4.6.0 in /usr/local/lib/python3.12/dist-packages (from ultralytics) (4.10.0)
Requirement already satisfied: pillow>=7.1.2 in /usr/local/lib/python3.12/dist-packages (from ultralytics) (11.3.0)
Requirement already satisfied: pyyaml>=5.3.1 in /usr/local/lib/python3.12/dist-packages (from ultralytics) (6.0.3)
Requirement already satisfied: requests>=2.23.0 in /usr/local/lib/python3.12/dist-packages (from ultralytics) (2.32.0)
Requirement already satisfied: scipy>=1.4.1 in /usr/local/lib/python3.12/dist-packages (from ultralytics) (1.16.0)
Requirement already satisfied: torch>=1.8.0 in /usr/local/lib/python3.12/dist-packages (from ultralytics) (2.9.0)
Requirement already satisfied: torchvision>=0.9.0 in /usr/local/lib/python3.12/dist-packages (from ultralytics) (0.19.0)
Requirement already satisfied: psutil>=5.8.0 in /usr/local/lib/python3.12/dist-packages (from ultralytics) (5.9.5)
Requirement already satisfied: polars>=0.20.0 in /usr/local/lib/python3.12/dist-packages (from ultralytics) (1.31.0)
Requirement already satisfied: ultralytics-thop>=2.0.18 in /usr/local/lib/python3.12/dist-packages (from ultralytics) (2.0.18)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=3.3.0) (1.3.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=3.3.0) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=3.3.0) (4.55.0)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=3.3.0) (1.4.7)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=3.3.0) (25.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=3.3.0) (3.2.0)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=3.3.0) (2.9.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.12/dist-packages (from requests>=2.23.0) (3.4.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.12/dist-packages (from requests>=2.23.0) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.12/dist-packages (from requests>=2.23.0) (2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.12/dist-packages (from requests>=2.23.0) (2025.11.12)
Requirement already satisfied: filelock in /usr/local/lib/python3.12/dist-packages (from torch>=1.8.0) (3.16.1)
Requirement already satisfied: typing-extensions>=4.10.0 in /usr/local/lib/python3.12/dist-packages (from torch>=1.8.0) (4.13.0)
Requirement already satisfied: setuptools in /usr/local/lib/python3.12/dist-packages (from torch>=1.8.0) (75.8.2)
Requirement already satisfied: sympy>=1.13.3 in /usr/local/lib/python3.12/dist-packages (from torch>=1.8.0) (1.14.0)
Requirement already satisfied: networkx>=2.5.1 in /usr/local/lib/python3.12/dist-packages (from torch>=1.8.0) (3.5)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.12/dist-packages (from torch>=1.8.0) (3.1.5)
Requirement already satisfied: fsspec>=0.8.5 in /usr/local/lib/python3.12/dist-packages (from torch>=1.8.0) (2025.10.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.7) (1.17.0)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.12/dist-packages (from sympy>=1.13.3) (1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.12/dist-packages (from jinja2) (3.0.2)
```

Reasoning: Now that `ultralytics` is installed, I will import the `YOLO` class and define the paths for the models and image files as specified in the subtask. This step consolidates the remaining setup requirements.

```
from ultralytics import YOLO

# Define model paths
trained_model_path = 'Trained_yolov8.pt'
kaggle_model_path = 'Kaggle_yolov8s.pt'

# Define image paths
image_paths = [
    'mango.jpg',
    'orange.jpg',
    'grapes.jpg',
```

```
'banana.jpg',
'strawberry.jpg',
'watermelon.jpg',
'apple.jpg',
'pineapple.jpg'
]

print("Ultralytics YOLO class imported. Model and image paths defined.")

Ultralytics YOLO class imported. Model and image paths defined.
```

Run Predictions with Trained Model

Subtask:

Load the Trained_yolov8.pt model using ultralytics and run predictions on all 8 image files. Store the results (e.g., detected objects, bounding boxes, confidence scores) and save the annotated images.

Reasoning: I will load the 'Trained_yolov8.pt' model and then iterate through the defined image paths, running predictions for each image and saving the results and annotated images as specified in the instructions.

```
trained_model = YOLO(trained_model_path)
trained_model_results = []

for image_path in image_paths:
    print(f"Running prediction on {image_path} with Trained YOLOv8 model...")
    results = trained_model.predict(source=image_path, save=True, conf=0.25)
    trained_model_results.append(results)
    print(f"Prediction for {image_path} completed.")

print("All predictions with Trained YOLOv8 model completed and results stored.")
```

Running prediction on mango.jpg with Trained YOLOv8 model...

image 1/1 /content/mango.jpg: 448x640 2 mangos, 361.0ms
Speed: 9.8ms preprocess, 361.0ms inference, 2.3ms postprocess per image at shape (1, 3, 448, 640)
Results saved to /content/runs/detect/predict5
Prediction for mango.jpg completed.
Running prediction on orange.jpg with Trained YOLOv8 model...

image 1/1 /content/orange.jpg: 448x640 2 oranges, 301.8ms
Speed: 9.2ms preprocess, 301.8ms inference, 1.4ms postprocess per image at shape (1, 3, 448, 640)
Results saved to /content/runs/detect/predict5
Prediction for orange.jpg completed.
Running prediction on grapes.jpg with Trained YOLOv8 model...

image 1/1 /content/grapes.jpg: 384x640 1 grape, 203.2ms
Speed: 6.4ms preprocess, 203.2ms inference, 0.9ms postprocess per image at shape (1, 3, 384, 640)
Results saved to /content/runs/detect/predict5
Prediction for grapes.jpg completed.
Running prediction on banana.jpg with Trained YOLOv8 model...

image 1/1 /content/banana.jpg: 576x640 1 banana, 306.3ms
Speed: 6.7ms preprocess, 306.3ms inference, 1.4ms postprocess per image at shape (1, 3, 576, 640)
Results saved to /content/runs/detect/predict5
Prediction for banana.jpg completed.
Running prediction on strawberry.jpg with Trained YOLOv8 model...

image 1/1 /content/strawberry.jpg: 448x640 2 strawberries, 386.7ms
Speed: 9.5ms preprocess, 386.7ms inference, 1.8ms postprocess per image at shape (1, 3, 448, 640)
Results saved to /content/runs/detect/predict5
Prediction for strawberry.jpg completed.
Running prediction on watermelon.jpg with Trained YOLOv8 model...

image 1/1 /content/watermelon.jpg: 448x640 1 watermelon, 428.6ms
Speed: 4.9ms preprocess, 428.6ms inference, 1.5ms postprocess per image at shape (1, 3, 448, 640)
Results saved to /content/runs/detect/predict5
Prediction for watermelon.jpg completed.
Running prediction on apple.jpg with Trained YOLOv8 model...

image 1/1 /content/apple.jpg: 640x640 2 apples, 471.2ms
Speed: 25.5ms preprocess, 471.2ms inference, 1.4ms postprocess per image at shape (1, 3, 640, 640)
Results saved to /content/runs/detect/predict5
Prediction for apple.jpg completed.
Running prediction on pineapple.jpg with Trained YOLOv8 model...

image 1/1 /content/pineapple.jpg: 448x640 4 pineapples, 315.2ms
Speed: 5.1ms preprocess, 315.2ms inference, 1.7ms postprocess per image at shape (1, 3, 448, 640)
Results saved to /content/runs/detect/predict5
Prediction for pineapple.jpg completed.
All predictions with Trained YOLOv8 model completed and results stored.

Run Predictions with Kaggle Model

Subtask:

Load the `Kaggle_yolov8s.pt` model using ultralytics and run predictions on all 8 image files. Store the results (e.g., detected objects, bounding boxes, confidence scores) and save the annotated images.

Instructions

1. Load the `Kaggle_yolov8s.pt` model using `YOLO()` and store it in a variable, for example, `kaggle_model`.
2. Iterate through each image path in the `image_paths` list.
3. For each image, run predictions using the `kaggle_model`'s `predict()` method. Make sure to save the results in a list, for example, `kaggle_model_results`.
4. Save the annotated image by specifying `save=True` in the `predict()` method arguments. The annotated images will be saved to a default directory, usually `'runs/detect/predict/'`.

Reasoning: Following the outlined subtask, I will now load the 'Kaggle_yolov8s.pt' model and iterate through the defined image paths, running predictions for each image and saving the results and annotated images. This mirrors the process for the trained model to prepare for comparison.

```
kaggle_model = YOLO(kaggle_model_path)
kaggle_model_results = []

for image_path in image_paths:
    print(f"Running prediction on {image_path} with Kaggle YOLOv8 model...")
    results = kaggle_model.predict(source=image_path, save=True, conf=0.25)
    kaggle_model_results.append(results)
    print(f"Prediction for {image_path} completed.")

print("All predictions with Kaggle YOLOv8 model completed and results stored.")
```

Running prediction on mango.jpg with Kaggle YOLOv8 model...

image 1/1 /content/mango.jpg: 448x640 2 apples, 564.7ms
 Speed: 3.9ms preprocess, 564.7ms inference, 3.1ms postprocess per image at shape (1, 3, 448, 640)
 Results saved to `/content/runs/detect/predict6`
 Prediction for mango.jpg completed.
 Running prediction on orange.jpg with Kaggle YOLOv8 model...

image 1/1 /content/orange.jpg: 448x640 2 oranges, 672.0ms
 Speed: 4.6ms preprocess, 672.0ms inference, 4.0ms postprocess per image at shape (1, 3, 448, 640)
 Results saved to `/content/runs/detect/predict6`
 Prediction for orange.jpg completed.
 Running prediction on grapes.jpg with Kaggle YOLOv8 model...

image 1/1 /content/grapes.jpg: 384x640 4 apples, 3 oranges, 789.4ms
 Speed: 3.8ms preprocess, 789.4ms inference, 1.8ms postprocess per image at shape (1, 3, 384, 640)
 Results saved to `/content/runs/detect/predict6`
 Prediction for grapes.jpg completed.
 Running prediction on banana.jpg with Kaggle YOLOv8 model...

image 1/1 /content/banana.jpg: 576x640 1 banana, 1263.2ms
 Speed: 6.8ms preprocess, 1263.2ms inference, 4.2ms postprocess per image at shape (1, 3, 576, 640)
 Results saved to `/content/runs/detect/predict6`
 Prediction for banana.jpg completed.
 Running prediction on strawberry.jpg with Kaggle YOLOv8 model...

image 1/1 /content/strawberry.jpg: 448x640 (no detections), 1140.3ms
 Speed: 9.1ms preprocess, 1140.3ms inference, 1.4ms postprocess per image at shape (1, 3, 448, 640)
 Results saved to `/content/runs/detect/predict6`
 Prediction for strawberry.jpg completed.
 Running prediction on watermelon.jpg with Kaggle YOLOv8 model...

image 1/1 /content/watermelon.jpg: 448x640 3 persons, 1547.6ms
 Speed: 9.9ms preprocess, 1547.6ms inference, 3.9ms postprocess per image at shape (1, 3, 448, 640)
 Results saved to `/content/runs/detect/predict6`
 Prediction for watermelon.jpg completed.
 Running prediction on apple.jpg with Kaggle YOLOv8 model...

image 1/1 /content/apple.jpg: 640x640 2 apples, 1819.3ms
 Speed: 13.6ms preprocess, 1819.3ms inference, 2.2ms postprocess per image at shape (1, 3, 640, 640)
 Results saved to `/content/runs/detect/predict6`
 Prediction for apple.jpg completed.
 Running prediction on pineapple.jpg with Kaggle YOLOv8 model...

image 1/1 /content/pineapple.jpg: 448x640 (no detections), 954.5ms
 Speed: 14.4ms preprocess, 954.5ms inference, 3.5ms postprocess per image at shape (1, 3, 448, 640)
 Results saved to `/content/runs/detect/predict6`
 Prediction for pineapple.jpg completed.
 All predictions with Kaggle YOLOv8 model completed and results stored.

Visualize Model Predictions

Subtask:

Display the original images alongside the predictions from both models (Trained_yolov8.pt and Kaggle_yolov8s.pt) for each image to allow for side-by-side visual comparison.

Reasoning: To visually compare the model predictions, I will import necessary libraries, then iterate through each image, load its original version, and the versions annotated by both models from their respective save directories, and display them in a 3-column subplot.

```
import matplotlib.pyplot as plt
from PIL import Image
import os

# Define the directories where annotated images are saved
trained_model_output_dir = 'runs/detect/predict'
kaggle_model_output_dir = 'runs/detect/predict2'

print("Starting visual comparison of model predictions...")

for i, image_path in enumerate(image_paths):
    # Extract filename from the path
    image_filename = os.path.basename(image_path)

    # Construct full paths for original and annotated images
    original_image_full_path = image_path
    trained_annotated_image_full_path = os.path.join(trained_model_output_dir, image_filename)
    kaggle_annotated_image_full_path = os.path.join(kaggle_model_output_dir, image_filename)

    # Create a figure with three subplots
    fig, axes = plt.subplots(1, 3, figsize=(18, 6))

    # Display Original Image
    try:
        original_img = Image.open(original_image_full_path)
        axes[0].imshow(original_img)
        axes[0].set_title('Original Image')
        axes[0].axis('off')
    except FileNotFoundError:
        axes[0].set_title('Original Image (Not Found)')
        axes[0].axis('off')
        print(f"Original image not found: {original_image_full_path}")

    # Display Trained Model's Annotated Image
    try:
        trained_annotated_img = Image.open(trained_annotated_image_full_path)
        axes[1].imshow(trained_annotated_img)
        axes[1].set_title('Trained Model Prediction')
        axes[1].axis('off')
    except FileNotFoundError:
        axes[1].set_title('Trained Model Prediction (Not Found)')
        axes[1].axis('off')
        print(f"Trained model annotated image not found: {trained_annotated_image_full_path}")

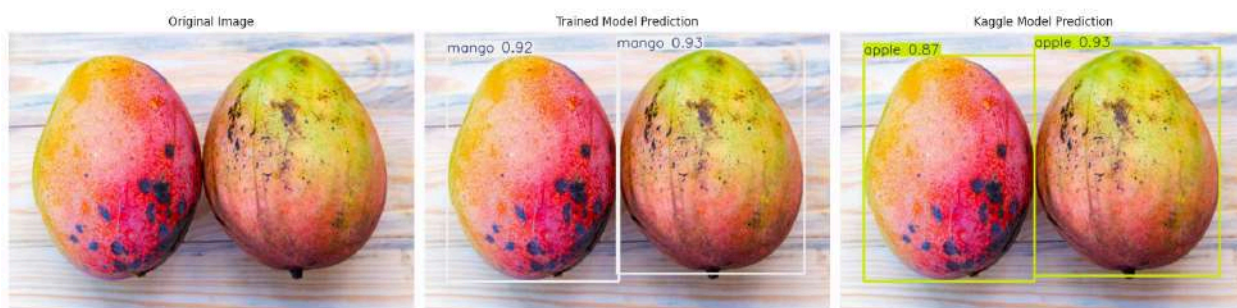
    # Display Kaggle Model's Annotated Image
    try:
        kaggle_annotated_img = Image.open(kaggle_annotated_image_full_path)
        axes[2].imshow(kaggle_annotated_img)
        axes[2].set_title('Kaggle Model Prediction')
        axes[2].axis('off')
    except FileNotFoundError:
        axes[2].set_title('Kaggle Model Prediction (Not Found)')
        axes[2].axis('off')
        print(f"Kaggle model annotated image not found: {kaggle_annotated_image_full_path}")

    plt.suptitle(f"Comparison for {image_filename}", fontsize=16)
    plt.tight_layout(rect=[0, 0.03, 1, 0.95]) # Adjust layout to prevent supitle overlap
    plt.show()

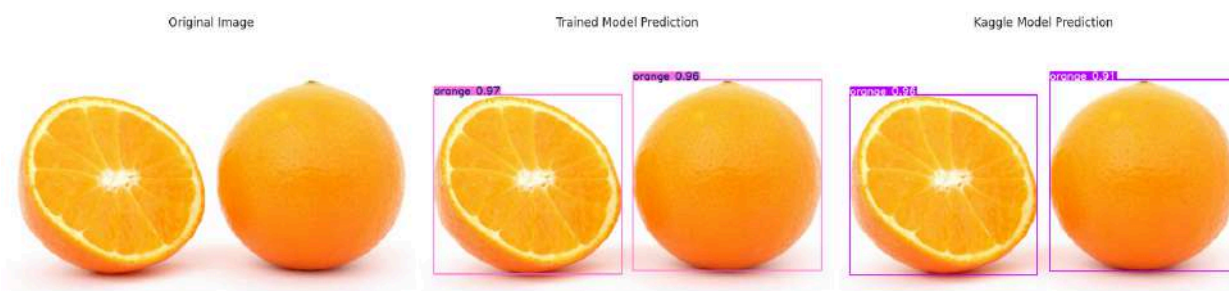
print("Visual comparison completed for all images.")
```


Starting visual comparison of model predictions...

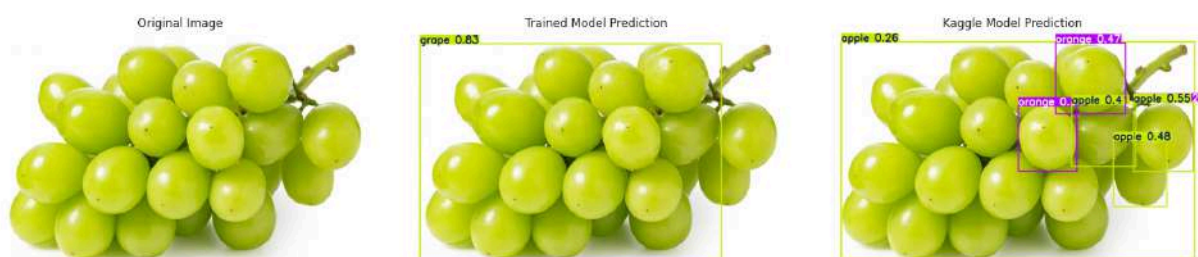
Comparison for mango.jpg



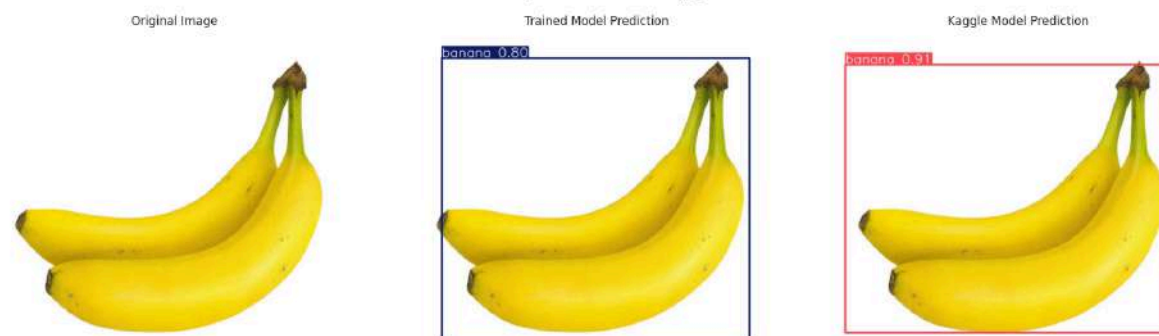
Comparison for orange.jpg



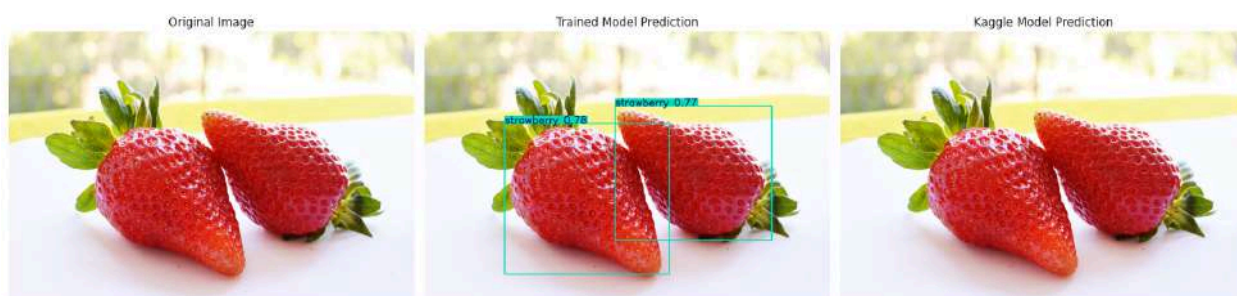
Comparison for grapes.jpg



Comparison for banana.jpg



Comparison for strawberry.jpg



Comparison for watermelon.jpg



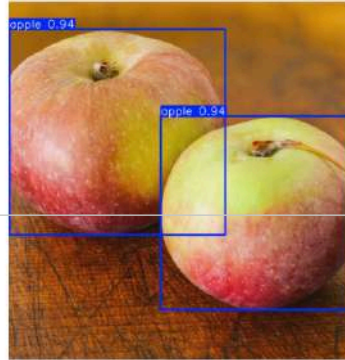


Comparison for apple.jpg

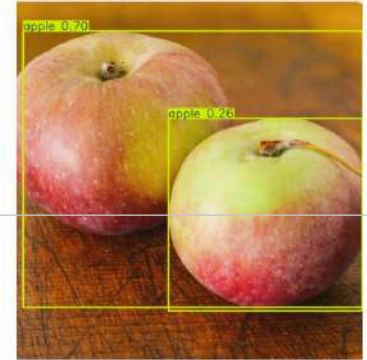
Original Image



Trained Model Prediction



Kaggle Model Prediction

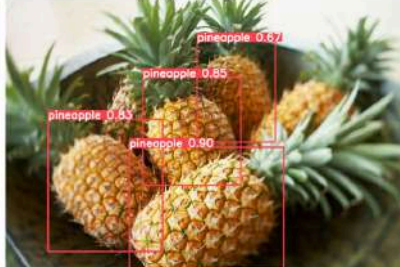


Comparison for pineapple.jpg

Original Image



Trained Model Prediction



Kaggle Model Prediction



Visual comparison completed for all images.

✓ Extract and Compare Metrics

Subtask:

Process the prediction results from both models. Extract and compare quantitative metrics such as the total number of detections per image, the types of objects detected, and average confidence scores for detected objects. Provide a framework (code snippet) for calculating F1-score and accuracy, explaining that ground truth bounding boxes and class labels for each image would be required to perform these calculations.

Reasoning: I will extract the quantitative metrics from the prediction results of both models as per the subtask instructions. This involves iterating through the results, extracting detection counts, object types, and average confidence scores, and storing them in lists of dictionaries.

```
import pandas as pd
from collections import Counter

trained_model_metrics = []
kaggle_model_metrics = []

print("Extracting metrics for Trained Model...")
for i, image_path in enumerate(image_paths):
    image_filename = os.path.basename(image_path)
    trained_results = trained_model_results[i][0] # Access the first (and likely only) result object

    # Trained Model Metrics
    if trained_results.bboxes and len(trained_results.bboxes) > 0:
        num_detections_trained = len(trained_results.bboxes)
        detected_classes_trained_ids = trained_results.bboxes.cls.tolist()
        detected_classes_trained_names = [trained_model.names[int(cls_id)] for cls_id in detected_classes_trained_ids]
        trained_class_counts = Counter(detected_classes_trained_names) # Store counts of each class
        confidence_scores_trained = trained_results.bboxes.conf.tolist()
        avg_confidence_trained = sum(confidence_scores_trained) / len(confidence_scores_trained)
    else:
        num_detections_trained = 0
        detected_classes_trained_names = []
        trained_class_counts = Counter()
        avg_confidence_trained = 0.0

    trained_model_metrics.append({
        'Image': image_filename,
        'Model': 'Trained',
        'Num Detections': num_detections_trained,
        'Detected Objects (Unique)': ', '.join(sorted(list(set(detected_classes_trained_names)))),
        'Detailed Class Counts': trained_class_counts, # New detailed counts
        'Average Confidence': avg_confidence_trained
    })

print("Extracting metrics for Kaggle Model...")
for i, image_path in enumerate(image_paths):
    image_filename = os.path.basename(image_path)
    kaggle_results = kaggle_model_results[i][0] # Access the first (and likely only) result object

    # Kaggle Model Metrics
    if kaggle_results.bboxes and len(kaggle_results.bboxes) > 0:
        num_detections_kaggle = len(kaggle_results.bboxes)
        detected_classes_kaggle_ids = kaggle_results.bboxes.cls.tolist()
        detected_classes_kaggle_names = [kaggle_model.names[int(cls_id)] for cls_id in detected_classes_kaggle_ids]
        kaggle_class_counts = Counter(detected_classes_kaggle_names) # Store counts of each class
        confidence_scores_kaggle = kaggle_results.bboxes.conf.tolist()
        avg_confidence_kaggle = sum(confidence_scores_kaggle) / len(confidence_scores_kaggle)
    else:
        num_detections_kaggle = 0
        detected_classes_kaggle_names = []
        kaggle_class_counts = Counter()
        avg_confidence_kaggle = 0.0

    kaggle_model_metrics.append({
        'Image': image_filename,
        'Model': 'Kaggle',
        'Num Detections': num_detections_kaggle,
        'Detected Objects (Unique)': ', '.join(sorted(list(set(detected_classes_kaggle_names)))),
        'Detailed Class Counts': kaggle_class_counts, # New detailed counts
        'Average Confidence': avg_confidence_kaggle
    })

# Convert to DataFrames and concatenate
trained_df = pd.DataFrame(trained_model_metrics)
kaggle_df = pd.DataFrame(kaggle_model_metrics)
```

```
kaggle_df = pd.DataFrame(kaggle_model_metrics)

comparison_df = pd.concat([trained_df, kaggle_df]).reset_index(drop=True)

print("Comparison of Extracted Metrics (Head):")
print(comparison_df.head(10))
```

```
Extracting metrics for Trained Model...
Extracting metrics for Kaggle Model...
Comparison of Extracted Metrics (Head):
```

	Image	Model	Num Detections	Detected Objects (Unique)
0	mango.jpg	Trained	2	mango
1	orange.jpg	Trained	2	orange
2	grapes.jpg	Trained	1	grape
3	banana.jpg	Trained	1	banana
4	strawberry.jpg	Trained	2	strawberry
5	watermelon.jpg	Trained	1	watermelon
6	apple.jpg	Trained	2	apple
7	pineapple.jpg	Trained	4	pineapple
8	mango.jpg	Kaggle	2	apple
9	orange.jpg	Kaggle	2	orange

	Detailed Class Counts	Average Confidence
0	{'mango': 2}	0.921372
1	{'orange': 2}	0.964910
2	{'grape': 1}	0.833267
3	{'banana': 1}	0.804425
4	{'strawberry': 2}	0.772422
5	{'watermelon': 1}	0.919901
6	{'apple': 2}	0.938955
7	{'pineapple': 4}	0.813859
8	{'apple': 2}	0.900074
9	{'orange': 2}	0.935845

Explanation on F1-score and Accuracy Calculation

To calculate robust evaluation metrics such as **F1-score** and **accuracy** for object detection models, **ground truth data** is indispensable. Ground truth data refers to the actual, correct labels and bounding box coordinates for every object present in each image.

Without ground truth data, we cannot determine:

- **True Positives (TP)**: Correctly detected objects with correct class labels and sufficient overlap with ground truth bounding boxes.
- **False Positives (FP)**: Detections that do not correspond to any ground truth object, or detections with incorrect class labels or poor bounding box overlap.
- **False Negatives (FN)**: Ground truth objects that the model failed to detect.
- **True Negatives (TN)**: This metric is generally not directly applicable in object detection in the same way it is for classification, as it would imply correctly identifying 'empty' regions, which isn't standard.

F1-score (which is the harmonic mean of precision and recall) and **accuracy** rely on these counts (TP, FP, FN). For instance, Precision = TP / (TP + FP) and Recall = TP / (TP + FN). Without knowing the true state of objects in the images, these values cannot be computed.

In this comparison, due to the absence of ground truth data, we are limited to comparing the models based on intrinsic metrics from their predictions, such as the number of detections, detected object types, and confidence scores, as well as visual inspection.

Generate Comparison Graphs

Subtask:

Create visualizations to compare the models, such as bar charts showing the number of detected objects per image for each model, or potentially confidence distributions if meaningful. This will provide a quantitative overview of their performance differences.

Reasoning: I will import `matplotlib.pyplot` and `seaborn`, then create two bar charts using `comparison_df` to visualize 'Num Detections' and 'Average Confidence' for both models across all images, as specified in the instructions.

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

print("Generating comparison graphs...")

# Set style for plots
sns.set_style("whitegrid")

# Create a figure with two subplots
fig, axes = plt.subplots(1, 2, figsize=(20, 7))
```

```
fig, axes = plt.subplots(1, 2, figsize=(20, 7))
```

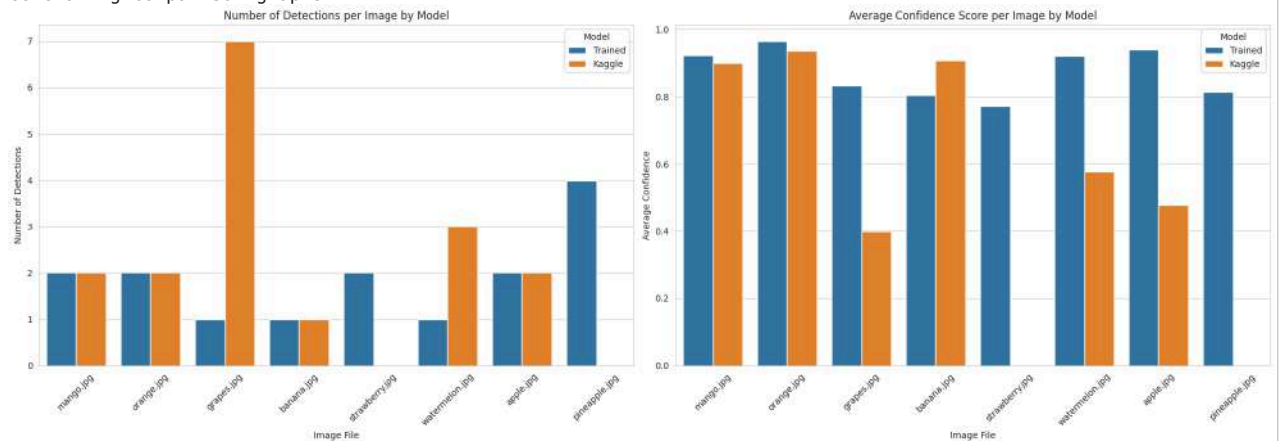
```
# Bar chart for Number of Detections
sns.barplot(x='Image', y='Num Detections', hue='Model', data=comparison_df, ax=axes[0])
axes[0].set_title('Number of Detections per Image by Model')
axes[0].set_xlabel('Image File')
axes[0].set_ylabel('Number of Detections')
axes[0].tick_params(axis='x', rotation=45) # Rotate x-axis labels for readability

# Bar chart for Average Confidence Scores
sns.barplot(x='Image', y='Average Confidence', hue='Model', data=comparison_df, ax=axes[1])
axes[1].set_title('Average Confidence Score per Image by Model')
axes[1].set_xlabel('Image File')
axes[1].set_ylabel('Average Confidence')
axes[1].tick_params(axis='x', rotation=45) # Rotate x-axis labels for readability

plt.tight_layout()
plt.show()

print("Comparison graphs generated.")
```

Generating comparison graphs...



Comparison graphs generated.

Reasoning: To visualize how 'wrong' the Kaggle model's detections were, I will compare its **Detailed Class Counts** against those of the Trained model (acting as a proxy for ground truth). I will categorize the discrepancies into 'Correctly Identified', 'Missed by Kaggle', and 'Kaggle False Positives/Misclassifications' per image and store them in a new DataFrame for plotting.

```
discrepancy_data = []

for image_filename in image_paths:
    trained_counts_row = comparison_df[(comparison_df['Image'] == image_filename) & (comparison_df['Model'] == 'Trained')]
    kaggle_counts_row = comparison_df[(comparison_df['Image'] == image_filename) & (comparison_df['Model'] == 'Kaggle')]

    trained_counts = trained_counts_row['Detailed Class Counts']
    kaggle_counts = kaggle_counts_row['Detailed Class Counts']

    correctly_identified = 0
    missed_by_kaggle = 0
    kaggle_false_positives = 0

    # Compare counts for classes detected by the Trained Model
    for cls, trained_count in trained_counts.items():
        kaggle_count = kaggle_counts.get(cls, 0)
        correctly_identified += min(trained_count, kaggle_count)
        if trained_count > kaggle_count:
            missed_by_kaggle += (trained_count - kaggle_count)

    # Identify false positives/misclassifications by Kaggle (objects Kaggle detected that Trained did not,
    # or over-counts for classes that Trained also detected)
    for cls, kaggle_count in kaggle_counts.items():
        trained_count = trained_counts.get(cls, 0)
        if kaggle_count > trained_count:
            kaggle_false_positives += (kaggle_count - trained_count)
```