Parking Space Detector

Introduction:

The objective of our project was to design a machine learning solution capable of detecting available and occupied parking spaces using computer vision. Parking availability detection has become increasingly important as cities are densely growing and people spend more time searching for a spot to park. A reliable automated solution with spot detection can reduce congestion, improve the flow of traffic, and better support city infrastructure.

We found this project to be interesting and relevant because parking spot detection blends together multiple machine learning and artificial intelligence skills. These skills being image processing, object detection, and classification models. The system developed in our project uses a YOLOv8 object detector along with a K-Nearest Neighbors classifier. This approach created a two stage workflow for the automation. YOLOv8 identifies the individual parking spaces within an image, and the KNN classifies the space as either occupied or empty based on the processed training data. Building the project required a full setup from environment setup, data extraction, model training, prediction, visualization, and evaluation.

We faced many challenges throughout the development of this project. A major challenge that we encountered was preparing our dataset. Although the dataset provided the images needed for training, it was a challenge to make it usable by both models. Variations in lightning, perspective, and image quality affected image preprocessing. These issues were addressed through uniform resizing and contrast enhancement to make the training data more consistent. Another challenge we encountered was tuning the models so that YOLO regions that are detected accurately so that the KNN can classify them with reliability. Iterative adjustments to our parameters helped improve the performance.

Dataset:

Our project uses the PKLot dataset, a large and widely used dataset for parking space occupancy detection. PKLot contains 12,417 parking lot images captured at 720p resolution. Images are also captured under varying weather conditions such as sunny, cloudy, and rainy. The images come from surveillance camera frames from multiple viewpoints to provide variation in perspective, angle, and distance. Each image includes many marked parking spaces, along with detailed annotations for each space. The PKLot dataset totals 695,899 labeled parking space instances, each being marked as occupied or empty. The size and variability in this dataset makes it well suited for training models where the goal is to generalize to any other real world parking lot.

The PKLot dataset served two major purposes for our project. The first being the full scene parking lot images being used in our detection stage, where YOLO was responsible for identifying the location of individual parking spaces within a larger image. The second being the dataset's annotated parking space regions being used to generate training data for the KNN. Based on the datasets labels, the code categorized the parking spaces into empty and occupied. This gave the classifier a standardized set of samples, despite the variation that occurs in the parking lot images.

Due to the annotations in the PKLot dataset, it was not necessary to curate or manually label any dataset for this project. Instead, our focus was on transforming the existing data to fit into the two stage pipeline. Overall, the variability, size, and detailed labeling of the PKLot dataset made it ideal for developing a parking space detection and classification machine learning system.

Evaluation Metrics:

Our project used several quantitative metrics to evaluate both stages of the parking detection system, the YOLOv8 detector and the KNN classifier.

For the YOLO detection model, the primary metrics were mAP50, mAP50-95, precision, and recall. These are standard metrics for an object detection task. The model was able to achieve strong results on the validation set, with an mAP50 of 0.9747, mAP50-95 of 0.7081, precision of 0.9388, and recall of 0.9551. These metric results indicate that YOLO was able to detect the parking spaces accurately with minimal incorrect detections. These metrics prove it to be reliable for providing high quality input for the KNN classification stage.

The KNN classifier evaluated the performance using accuracy, precision, recall, F1 score, and a confusion matrix. These performance metrics are appropriate for binary classification problems, and we are classifying between an open parking space and an occupied space. The model had an accuracy of 0.8340. For the free spaces there was a precision of 0.74 with a recall of 1. For occupied spaces the precision was 1.00 with a recall score of 0.68. The F1 scores were 0.85 for free spaces and 0.81 for occupied spaces respectively. The confusion matrix is as follows:

[[259 0]

[89 188]]

The confusion matrix shows that the KNN classifier was able to correctly identify all free spaces, having zero false positives. It has difficulty detecting the occupied spaces, having 89 false negatives.

Baseline Model:

For our project the baseline model was the K Nearest Neighbors (KNN) classifier, which was used to provide a simple and interpretable benchmark for predicted parking space occupancy. Each full parking lot image was cropped to one parking space patched and converted

to a 32x32 greyscale image and also flattened into a 1024 dimensional feature vector. The KNN model is classified using k = 5 neighbors with distance weighting. This allowed for a straightforward method to gauge how well a traditional machine learning model performs on this task without complex training. The baseline achieved a validation accuracy of 83.40% with a stronger performance on open parking spaces and a good performance on occupied spaces.

Details of Method:

Our project uses a two stage pipeline to detect and classify parking spaces in the PKLot dataset. The YOLOv8 detection model has an input of a full 720p parking lot images. It outputs bounding boxes for each parking space. The KNN occupancy classifier has input of cropped patches of individual parking spaces, extracted using the YOLO labels. It outputs a binary prediction for each patch, an open space (0) or an occupied space (1). The idea behind this design for our method is to combine the strengths of an object detector (YOLOv8) with a simple baseline classifier (KNN) that operates on the image patches. In short, YOLO handles where the parking spaces are, and the KNN focuses on whether a specific space is empty or occupied.

Here is the implementation of the YOLOv8 training and validation:

```
# Train the model
trainingResults = modelToTrain.train(
    data=configPath,
    epochs=trainingConfig['epochs'],
    batch=trainingConfig['batch'],
    imgsz=trainingConfig['imgsz'],
    patience=trainingConfig['patience'],
    optimizer=trainingConfig['optimizer'],
    lr0=trainingConfig['lr0'],
    device=trainingConfig['device'],
    project=datasetDirectory,
    name='parking_detector',
    exist_ok=True,
    save=True,
    verbose=True
)

print("\nTraining completed!")
print(f"Best model saved to: {datasetDirectory}/parking_detector/weights/best.pt")

return trainingResults
```

```python
# Train a YOLO detection model using the provided configuration and checkpoint directory.
def trainYoloModel(configPath, trainingConfig, checkpointDir):

    # Detailed overview of `trainYoloModel`:
    # Overall training pipeline inside this function:
    # 1) Load the YOLO model architecture and (optionally) pretrained weights from the model path.
    # 2) Attach a custom `CheckpointCallback` so that training periodically saves `.pt` snapshots.
    # 3) Pass dataset config + hyperparameters (epochs, batch size, image size, etc.) to YOLO's `train` API.
    # 4) Let Ultralytics handle the low-level training loop (forward/backward passes, optimizer steps).
    # 5) Save the final/best model weights into the provided checkpoint directory for later evaluation.

    """
    Train YOLO model with custom checkpoint callback.

    Args:
        configPath (str): Path to dataset YAML configuration
        trainingConfig (dict): Training hyperparameters
        checkpointDir (str): Directory for checkpoint storage

    Returns:
        results: Training results object
    """
    print("=" * 70)
    print("Starting Model Training")
    print("=" * 70)

    # Initialize model
    modelToTrain = YOLO(trainingConfig['model'])

    # Create checkpoint callback
    checkpointCallback = CheckpointCallback(
        checkpointDirectory=checkpointDir,
        saveInterval=trainingConfig['checkpoint_interval']
    )

    # Add callback using the correct ultralytics method
    modelToTrain.add_callback('on_train_epoch_end', checkpointCallback)
```

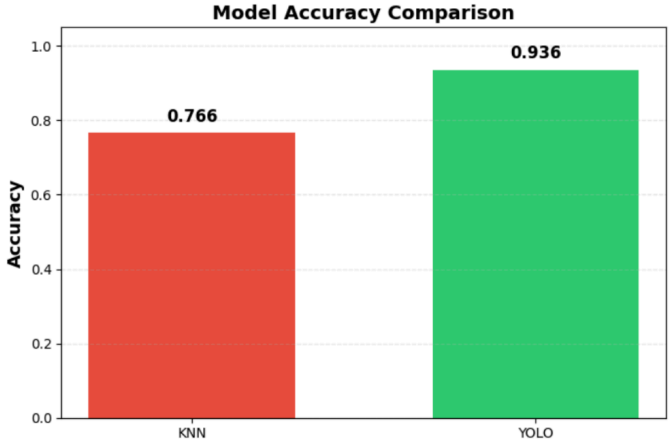
```python
# Evaluate a trained YOLO model on test images and compute accuracy-style metrics.
def evaluateYoloModel(modelPath, dataConfigPath):

    # Detailed overview of `evaluateYoloModel`:
    # What YOLO evaluation does here:
    # 1) Load the best or chosen YOLO checkpoint from disk.
    # 2) Run inference on all images in the test split using the chosen confidence / IoU thresholds.
    # 3) For each image, compare YOLO's predicted occupancy against the ground-truth labels.
    # 4) Accumulate simple accuracy statistics (e.g., correct vs total slots) into a summary dict.
    # 5) Return those metrics so they can be compared to the baseline KNN performance.

    """
    Evaluate trained model on validation set.

    Args:
        modelPath (str): Path to trained model weights
        dataConfigPath (str): Path to dataset configuration

    Returns:
        metrics: Validation metrics
    """
    print("=" * 70)
    print("YOLO Model Evaluation")
    print("=" * 70)

    # Load trained model
    evaluationModel = YOLO(modelPath)

    # Run validation
    validationMetrics = evaluationModel.val(
        data=dataConfigPath,
        split='val',
        save_json=True,
        save_hybrid=False
    )

    # Display results
    print("\nValidation Results:")
    print(f"mAP50: {validationMetrics.box.map50:.4f}")
    print(f"mAP50-95: {validationMetrics.box.map:.4f}")
    print(f"Precision: {validationMetrics.box.mp:.4f}")
    print(f"Recall: {validationMetrics.box.mr:.4f}")

    return validationMetrics
```

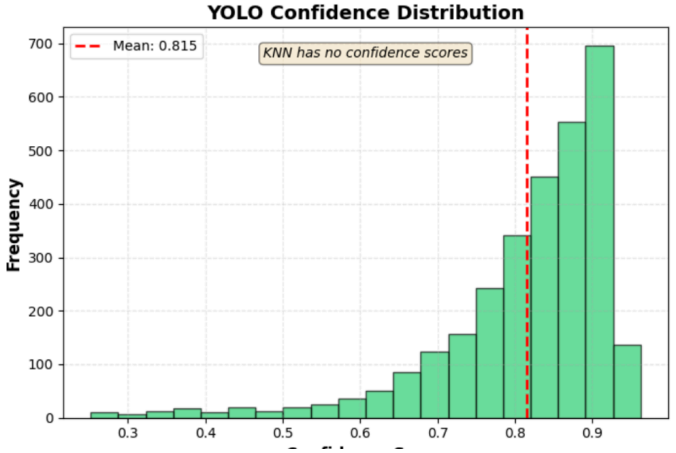
In both stages of the pipeline, the YOLOv8 detection and the KNN classification, we used carefully selected hyperparameters to balance performance, speed, and practicality. For the YOLOv8 model the project used the yolov8n architecture, which was chosen because it offers fast training while benefiting from pretrained weights. Training was limited to 1 epoch to keep the compute time manageable, with a batch size of 16 an image size of 1024x1024. This preserved enough important image detail while not exceeding memory limits. The model was optimized using Adam with the learning rate being 0.001.

For the KNN classifier the preprocessing step cropped parking space patches into a 32x32 grey image, making 1024 dimensional vectors. To keep training efficient while capturing the variability in the dataset, extraction was limited to 2000 training samples and 500 validation samples. The final model used k = 5 neighbors with distance weighting applied so that closer neighbors had a greater influence on the prediction. The ball tree algorithm was used for the nearest neighbor lookup due to its efficiency. These hyperparameters ensured that the workflow remained efficient while producing strong results.

Results:

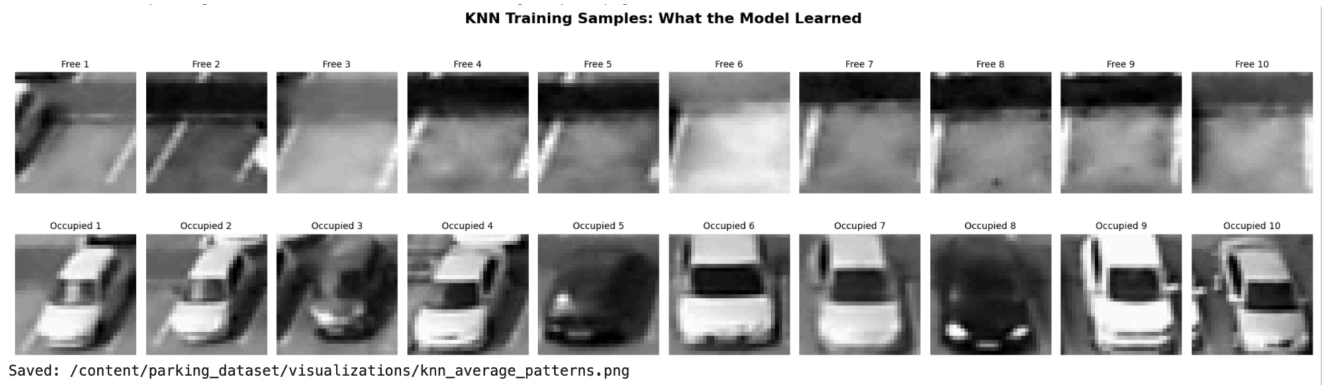
Model	Accuracy	Precision	Recall	F1 Score	Additional Metrics
KNN Baseline	83.40%	Free: 0.74 Occupied: 1.00	Free: 1.00 Occupied: 0.68	Free: 0.85 Occupied: 0.81	Confusion Matrix: [[259 0] [89 188]]
YOLOv8 Detection	N/A	0.9388	0.9551	N/A	mAP50: 0.9747 mAP50-95: 0.7081

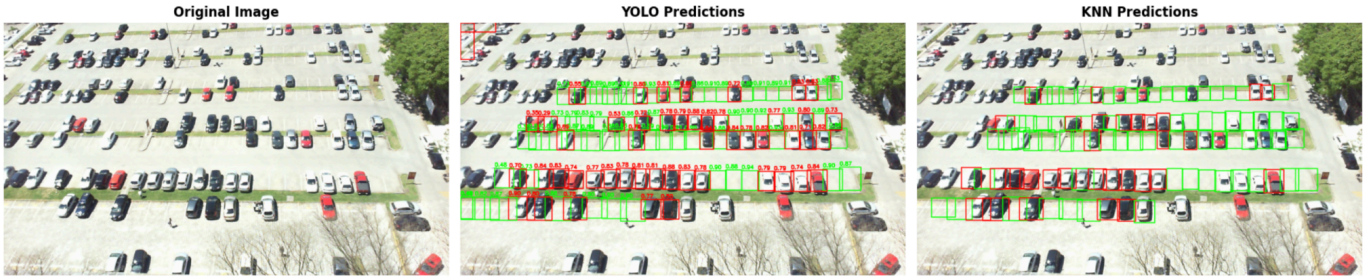




The results show our two stage approach using YOLOv8 for detection and the KNN for classification. This approach worked significantly better than the baseline model alone. The KNN classifier achieved an accuracy of 83.40% with a strong precision and a weaker recall, meaning it correctly labeled occupied spaces when it predicted them, but often failed to detect them. In contrast, the YOLOv8 model demonstrated a much stronger performance, having a precision of 0.9388, recall of 0.9551, and an mAP50 of 0.9747, which indicates high reliability at

identifying parking space regions and distinguishing between free and occupied slots.





Compared to the baseline model, the YOLO performed noticeably better. KNN struggled with occupied space recall because it relied only on greyscale image patches, while YOLO used deeper visual features and full scene context. This allowed YOLO to handle lighting changes, shadows, and different vehicle appearances more effectively, resulting in less missed detections.

With more time, several improvements could be made to further strengthen the system. This could be training YOLO for more epochs, replacing KNN with a small CNN, or if possible using more training samples. It is likely that taking these steps and implementing these changes would increase the accuracy and improve generalization across different parking lots under different conditions.

Future Improvements and Lessons Learned:

Future improvements could include training YOLO for more epochs, experimenting with larger model variants, and replacing the KNN with a CNN to have stronger predictions for parking spot occupancy. Using more of the PKLot dataset or adding augmented data would also aid in generalizing better across different parking environments.

A key lesson learned was the importance of preprocessing and consistent image formatting, since small differences in the lighting or cropping of an image can have a significant impact on the performance. Another lesson was how much more effective a deep learning model is compared to a standard baseline like KNN when dealing with real world images that have visual variability. Overall, this project reinforced the value of building pipelines and evaluating each stage carefully.