Parking Spot Detection and Management System

This project aims to create a parking spot detection and management system that detects available parking spots and displays the information on a website. The system consists of a Python Flask website, a detection module using YOLOv5, and several other modules for data collection, correction, and model training and evaluation.

Overview

1. **Python Flask Website**: A web application that displays available parking spots in different locations using Flask and Jinja2.
2. **Detection Module**:

The Detection Module is a Python-based module that utilizes the YOLOv5 object detection model to process live camera feeds and detect parking spots. It is implemented in the `ParkingLotDetector` class, which includes the following attributes and methods:

Attributes:

- `detection_model`: The YOLOv5 object detection model.
- `cap`: VideoCapture object to capture video from camera or video file.
- `fps_start`: Tick count to measure the frame rate.
- `frame_count`: Number of processed frames.
- `colorStates`: Tuple of colors used to visualize the parking spot states.
- `strStates`: Tuple of string labels representing parking spot states.
- `readyToSend`: Flag indicating if the data is ready to be sent to the API.
- `proccess`: Flag indicating if the module should continue processing.
- `frameTosend`: The frame to be sent to the API.
- `parkings`: Dictionary containing information about the parking spots.
- `parkingsSendCopy`: A copy of the parking spots dictionary for sending to the API.
- `currentParkingID`: The ID of the current parking spot being processed.

Methods:

- `__init__`: Initializes the ParkingLotDetector with the specified model, video source, and camera settings.
- `getLocalisation`: Retrieves the geolocation information of the user.
- `prepare_parkings_data`: Prepares the parking spots data to be sent to the API.
- `update_API`: Sends the parking spots data to the API.
- `detect_frame`: Detects parking spots in a video frame.
- `get_remote_image`: Retrieves an image from a remote web source.
- `runRemoteSource`: Main loop that processes the remote video feed and detects parking spots.

This module is responsible for detecting parking spots in real-time by analyzing video frames from live camera feeds or video files. It processes each frame, identifies the parking spots, and sends the updated parking spot data to a web API for further analysis or display.

3. **Correction Module**:

The Correction Module is a Python-based module that corrects the detected parking spots by comparing them to the ground truth annotations. It is implemented in the `CorrectionModule` class, which includes the following attributes and methods:

Attributes:

- `images_dir`: The directory containing the input images.
- `annotations_dir`: The directory containing the ground truth annotations.
- `labels_dir`: The directory containing the labels (detected parking spots).
- `max_width`: The maximum width of the images.
- `max_height`: The maximum height of the images.

Methods:

- `__init__`: Initializes the CorrectionModule with the specified directories and image dimensions.
- `iou`: Calculates the Intersection over Union (IoU) between two bounding boxes.
- `distance`: Calculates the Euclidean distance between the centers of two bounding boxes.
- `inclusion_percentage`: Calculates the percentage of one bounding box included in another.
- `max_touching_distance`: Calculates the maximum distance between the centers of two touching bounding boxes.
- `analyze_boxes`: Analyzes two bounding boxes and returns a dictionary with the analysis results.
- `load_image`: Loads an image from a given path.
- `load_images`: Loads all images for a specific camera ID.
- `load_annotations`: Loads ground truth annotations for a specific camera ID.
- `load_labels`: Loads labels (detected parking spots) for a given image name.
- `compare_correction`: Compares the detected parking spots to the ground truth annotations and returns matched and unmatched predictions.
- `convert_to_yolo_format`: Converts corrected parking spot coordinates to YOLO format and saves them to a label file.
- `run_analysis`: Runs the correction analysis for a specific camera ID and saves the corrected results.

The Correction Module helps improve the accuracy of the parking spot detection by comparing the detected parking spots to the ground truth annotations. It calculates various metrics such as IoU, distance, and inclusion percentage to determine if a prediction is correct or not. The corrected parking spot coordinates are then converted to YOLO format and saved to label files for further use. 4. **Training and Evaluation Module**: Trains the model using the corrected data and evaluates its performance. If the new model performs better, it is sent to production.

Parking Detection Project Workflow

1. **Camera Feed**: Obtain live camera feed from various parking locations.
2. **Detection Module**: 2. **Detection Module**: Process the camera feed using YOLOv5 to detect available parking spots.
3. **Data Collection**: Collect the detected parking spot information and save it to the PostgreSQL database.
4. **Data Correction**: Correct the collected data, if necessary, to improve the accuracy of the information.
5. **Model Training**: Train the YOLOv5 model using the corrected data to improve detection accuracy.
6. **Model Evaluation**: Evaluate the model's performance and deploy the new model if it performs better.

7. **Web Application**: Display the available parking spot information on a Flask web application for users.

Modules, Attributes, and Methods

Module	Attributes	Methods
DetectionModule	- detection_model - cap - fps_start - frame_count - colorStates - strStates - readyToSend - proccess - frameTosend - parkings - parkingsSendCopy - currentParkingID	- **init**() - getLocalisation() - prepare_parkings_data() - update_API() - detect_parking_spots() - get_remote_image() - runRemoteSource()
CorrectionModule	- images_dir - annotations_dir - labels_dir - max_width - max_height	- **init**() - iou() - distance() - inclusion_percentage() - max_touching_distance() - analyze_boxes() - load_image() - load_images() - load_annotations() - load_labels() - compare_correction() - convert_to_yolo_format() - run_analysis()
ModelTraining	- training_data - evaluation_data	- train_model() - evaluate_model()
WebApplication	- flask_app - api_endpoint	- display_data()

DevOps Implementation

To implement a DevOps approach for this project, we will use the following tools and methodologies:

1. Version Control (Git & GitHub Actions)

- Use Git for version control and collaboration.
- Use GitHub as the remote repository and leverage GitHub Actions for CI/CD.

2. Configuration Management (YANG, YAML, XML, Ansible, Chef)

- Use YAML for most configuration files (e.g., Ansible playbooks, GitHub Actions workflows).
- Use XML if you have specific requirements for XML-based configurations.
- Use Ansible and Chef for configuration management and infrastructure provisioning.

3. Containerization (Docker)

- Containerize your applications and services using Docker for consistency across environments.

4. Web Application (Flask, Jinja2, PostgreSQL)

- Develop a Flask web application with Jinja2 for templating.
- Use PostgreSQL as the database for storing parking spot information and other data.

5. Development and Deployment (Red Hat Linux, Networking, ITIL)

- Develop and deploy your applications on Red Hat Linux.
- Follow ITIL best practices for IT service management.
- Ensure proper networking configurations for your services.

6. Moving to Azure

- When you're ready to move your project to Azure, you can leverage Azure services to simplify your DevOps workflow:
    - Azure Repos for Git-based version control.
    - Azure Pipelines for CI/CD.
    - Azure Container Registry for storing Docker images.
    - Azure Kubernetes Service (AKS) or Azure App Service for container orchestration and deployment.
    - Azure Database for PostgreSQL for managed PostgreSQL services.
    - Azure Monitor for monitoring and logging.

Getting Started

1. Clone the repository:

bash git clone <repository_url>

2. Install the required dependencies:

bash pip install -r requirements.txt

3. Run the Flask application:

bash export FLASK_APP=app.py flask run

4. Navigate to `http://localhost:5000` in your web browser to view the application.

Contributing

1. Create a new branch for your feature or bug fix:

```
git checkout -b feature/new-feature
```

2. Make your changes and commit them:

```
git add .
git commit -m "Add new feature"
```

3. Push your changes to the remote repository:

```
git push origin feature/new-feature
```