International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 09 Issue: 05 | May - 2025 SJIF Rating: 8.586 ISSN: 2582-3930

Tree Enumeration Using Image Analytics

Amal V, Shaik Md Asim, Ravi Teja, Lakshmi Swaroop, Dr Nagaraj SR
B.Tech, Computer Science and Engineering
Presidency University, Bangalore
May 2025

ABSTRACT--Accurate tree enumeration is essential for ecological monitoring, urban planning, and forest management. Traditionally, this task has been carried out manually, which is not only time-consuming but also prone to human error. This project aims to simplify and automate the process of tree counting using image analytics. By leveraging a deep learning-based object detection model—YOLOv8—we trained the system to detect and count trees from aerial and landscape images. The approach involves curating a custom dataset, annotating it using tools like Roboflow, and training the model on Google Colab. A simple web interface was developed using Flask to allow users to upload an image and receive real-time results showing the number and type of trees detected. The model performed well on various test images, showing a high detection accuracy. This system not only reduces manual effort but also provides a scalable and efficient solution for large-scale environmental data collection

Keywords: Tree enumeration, Image analytics, YOLOv8, YOLOv9, YOLOv10, Deep learning, Object detection, Streamlit, Python, OpenCV, TensorFlow, Environmental monitoring, Forest management.

I. INTRODUCTION

Tree enumeration plays a vital role in environmental conservation, urban development, and biodiversity studies. It involves identifying and counting trees in a particular region, which helps in monitoring forest cover, planning green spaces, and evaluating the ecological impact of urbanization. Traditionally, this process has been carried out through manual surveys or the use of GPS-enabled devices, both of which are time-intensive and often inaccurate, especially in large or dense forest areas. With the advancement of computer vision and deep learning, image-based analysis has emerged as a powerful tool to automate and enhance the accuracy of tree enumeration. In this project, we leverage image analytics techniques using a custom-trained YOLOv8 model to detect and count trees from aerial and landscape images.

Our system uses annotated datasets prepared via Roboflow, and the model is trained on Google Colab for optimized performance. To make the system user-friendly, a web interface is developed using Flask, allowing users to upload images and view detection results with tree count and classification.

This automation not only speeds up the enumeration process but also ensures consistency and scalability. It can be particularly beneficial for forestry departments, environmental researchers, and smart city planners. In the long run, such solutions can contribute to better ecosystem management and climate change monitoring by providing timely and accurate data.

II. PROBLEM STATEMENT AND DOMAIN OVERVIEW

Trees are an integral part of any urban environment, contributing to ecological balance, air purification, temperature regulation, and overall well-being. However, with rapid urbanization and shrinking green spaces, it has become increasingly important to monitor and manage tree cover effectively. Traditionally, tree enumeration—counting and cataloging trees—has relied on manual field surveys. While accurate, these methods are highly resource-intensive, time-consuming, and impractical for covering large areas, especially in densely populated cities. Moreover, human-based surveys are prone to inconsistencies and errors, particularly when dealing with vast terrains or limited manpower.

This project addresses the challenge of efficiently identifying and counting trees by applying image analytics and machine learning techniques. The aim is to automate the tree enumeration process using aerial or satellite images and computer vision algorithms. By leveraging advancements in deep learning, particularly Convolutional Neural Networks (CNNs), the system is trained to detect and classify tree objects from images with high accuracy. This not only speeds up the enumeration process but also enhances consistency and scalability, making it suitable for government agencies, urban planners, and environmental organizations.

The domain of this project lies at the intersection of computer vision, environmental monitoring, and smart city planning.

Using Python and machine learning frameworks like TensorFlow and OpenCV, the system processes visual data to extract meaningful insights about vegetation patterns. The broader goal is to contribute to sustainable urban development by enabling data-driven decisions around tree plantation, deforestation control, and resource allocation. In summary, this project falls within the domain of image analytics for environmental applications, with a specific focus on automating tree enumeration to support greener, smarter cities.

III. OBJECTIVES

The primary objective of this project is to design and implement an automated system capable of accurately identifying and enumerating trees using image analytics. With the growing need for efficient and scalable environmental monitoring solutions, this project aims to bridge the gap between traditional survey methods and modern computational techniques by utilizing image-based machine learning models. The system is intended to be robust, efficient, and adaptable across different types of landscapes and tree densities.Below are the detailed objectives of this work:

1. **Develop an Image-Based Tree Detection Model:** The foremost goal is to construct a model that can detect trees in aerial or satellite imagery with high precision. This involves selecting appropriate datasets, preprocessing images, and applying deep learning algorithms—especially Convolutional Neural Networks (CNNs)—to train the model for reliable object detection.
2. **Enable Accurate Tree Enumeration:** The project seeks to automate the process of counting trees in an image. The system should be able to distinguish individual trees, even in densely vegetated regions, thereby enabling accurate enumeration. This reduces the reliance on manual surveys, saving time and manpower while minimizing errors.
3. **Optimize the Model for Real-Time Use:** Another core objective is to ensure that the model runs efficiently and can process images quickly enough to be useful in real-world scenarios. The solution should be scalable so it can handle large datasets and process high-resolution images without significant lag or computational overhead.
4. **Ensure Versatility Across Diverse Terrains:** Trees vary significantly in shape, size, and appearance across different geographies. Therefore, the system must be trained to recognize various tree types and adapt to diverse environmental settings such as urban landscapes, semi-urban areas, and natural forests.
5. **Promote Environmental Data Collection and Urban Planning:** A long-term objective of this project is to provide a tool that assists city administrators, urban planners, and environmental agencies in tracking tree coverage, planning green spaces, and implementing sustainability programs. Automated tree enumeration can serve as a key input for urban development projects and environmental conservation efforts.
6. **Validate and Analyze Performance Metrics:** The model's performance must be evaluated using standard metrics such as accuracy, precision, recall, and F1-score. This helps in quantifying the effectiveness of the detection and enumeration processes and identifying areas for further improvement.
7. **Create a User-Friendly Workflow:** While technical accuracy is critical, ease of use is equally important. One of the project's aims is to package the workflow in a way that can be adopted by non-technical stakeholders, possibly through a simple GUI or script-based interface.

In essence, this project focuses on applying computer vision and deep learning to solve a real-world environmental problem, with the ultimate goal of supporting smart, sustainable cities through accurate, automated tree monitoring.

IV.TECHNICAL OVERVIEW & ARCHITECTURE

The Tree Enumeration Using Image Analytics system is built using a modular architecture that separates the frontend, backend, and model inference layers. This separation ensures better maintainability, scalability, and adaptability of the platform for different use cases and environments. The solution combines classical web development with deep learning to create a streamlined process for analyzing drone-captured or aerial imagery to detect and count trees. At a high level, the system takes in high-resolution images—captured either through drones, satellites, or any aerial imaging source—and processes them using deep learning techniques to detect individual trees and count them. The core pipeline involves several stages: image acquisition, preprocessing, model training, object detection, result post-processing, and visualization.

The entire pipeline is developed using Python, leveraging key libraries like TensorFlow, Keras, OpenCV, and NumPy. For object detection, the project utilizes the YOLO (You Only Look Once) architecture, specifically trained to recognize tree shapes and patterns.

The system architecture is divided into two major components: the frontend and the backend. These components work in coordination to handle the entire flow from image input to object detection output. The backend also integrates a pre-trained YOLOv5 (You Only Look Once version 5) model that serves as the core of the detection logic, while the frontend focuses on presenting the interaction layer to the user.

IV.I Frontend Development

The frontend of the application is built using HTML, CSS, and JavaScript, designed to be lightweight and highly responsive. Users can upload aerial images through a clean interface, which then triggers the detection process. The interface also handles the rendering of the output image with bounding boxes and the final count of detected trees. The design prioritizes clarity and accessibility, especially since users may not have technical expertise in deep learning or image analytics.

JavaScript manages the interactivity, including handling file uploads, calling backend APIs, and dynamically displaying the processed results. CSS is used to ensure that the layout adapts well to different screen sizes. Where needed, frontend frameworks such as Bootstrap can be employed to enhance responsiveness and styling, although the overall UI is intentionally kept minimal to avoid overwhelming the user. Below are the listed Frontend technologies:

- **HTML:** The structural foundation of the web-based frontend is built using HTML (HyperText Markup Language). This markup language is used to define the overall layout of the user interface, including elements such as the image upload section, response display area, and interface buttons. HTML provides the semantic framework upon which all other frontend components rely.
- **CSS:** To style and format the user interface, CSS (Cascading Style Sheets) is employed. It enhances the visual presentation of the HTML structure, ensuring the platform is visually appealing, responsive, and easy to use. CSS helps in adjusting the layout for various screen sizes, maintaining consistency in font styles, margins, paddings, and other visual elements, thus improving user experience across devices.
- **JavaScript:** The application's interactivity is primarily driven by JavaScript, which handles client-side scripting. JavaScript is used to enable real-time actions such as image selection, form validation, sending API requests to the backend, and rendering the processed output. It acts as the communication bridge between the user interface and the backend service, facilitating asynchronous behavior through AJAX or Fetch API**s.**

IV.II Backend Development

The backend is built using Python, leveraging the Flask web framework to manage server-side logic and API communication. When a user uploads an image through the frontend, Flask receives the request and initiates the detection pipeline. The image is first preprocessed, resized, normalized, and formatted and then passed into a deep learning model based on YOLOv5, which has been trained on a custom dataset of aerial images annotated with tree positions.

YOLOv5 is chosen for its efficiency and high performance in object detection tasks. It processes the image in a single forward pass, identifying multiple trees and their locations in near real-time. After detection, post-processing techniques such as non-maximum suppression (NMS) are applied to eliminate redundant bounding boxes. The final output includes a visual representation with bounding boxes drawn around each detected tree and a numeric count of total detections. Libraries such as OpenCV and NumPy are used in this phase to handle image manipulation and array operations. The backend then returns the result as a JSON response, which is received by the frontend and rendered back to the user. Below is a breakdown of the architecture and technologies used for Backend.

- **Python**: The core processing logic, including model loading and image analysis, is handled using Python on the backend. Python is selected for its rich ecosystem of libraries and ease of integrating deep learning models. It provides the flexibility to write clean, maintainable code while efficiently managing resource-heavy image operations and model inference.
- **Flask:** To implement the web server and create API endpoints, Flask, a lightweight web framework in Python, is used. Flask acts as the intermediary between the frontend and the deep learning model. When a user uploads an image, Flask receives the request, invokes the detection logic, and then returns the output (such as tree count and processed image) back to the frontend. Flask's simplicity and modular design allow for quick development and customization.
- **YOLOv5:** At the heart of the detection system lies YOLOv5 (You Only Look Once, version 5), a highly efficient deep learning model known for real-time object detection capabilities. YOLOv5 takes the uploaded aerial image as input and processes it through a convolutional neural network (CNN) to identify and localize trees. This model was trained on a dataset of annotated tree images and is capable of recognizing tree features even at varying scales and angles. Its balance between speed and accuracy makes it ideal for this type of real-world, large-image detection task.
- **OpenCV:** To handle image processing operations, OpenCV (Open Source Computer Vision Library) is integrated within the backend. OpenCV is used for preprocessing steps such as image resizing, format conversion, and overlaying bounding boxes on the detection results. Its optimized routines help reduce processing time and improve compatibility with the YOLOv5 output format.
- **NumPy**: The application also utilizes NumPy, a fundamental Python library for numerical computations. NumPy supports the manipulation of image arrays and facilitates interaction between OpenCV and YOLOv5 outputs. It is particularly useful during model input preparation and

output decoding stages, where performance and matrix operations are critical.

- **RESTful API:** The communication between the frontend and backend is established via a RESTful API protocol. This architecture allows the frontend to send image files using HTTP POST requests and receive structured data such as bounding box coordinates and tree count in return. REST ensures that the system remains stateless, scalable, and simple to integrate with additional tools or dashboards if needed in the future.
- **Model Weights and Checkpoints:** The YOLOv5 model operates based on pre-trained weights that are stored locally on the server. These weights were obtained by training on a custom dataset consisting of labeled tree images. During inference, these checkpoints are loaded dynamically to ensure faster processing and high detection accuracy.

IV.III Data Handling and Communication

The system does not require long-term storage of images or user sessions, making it suitable for stateless operation. Images are temporarily handled in memory or in a cache directory during processing The communication between frontend and backend is RESTful, with HTTP POST requests used to transmit uploaded images. The backend responds with processed data including detection confidence, coordinates of detected trees, and overall count, which the frontend then visualizes accordingly.

IV.IV Scalability and Deployment Considerations

The platform is designed with future scalability in mind. The backend system, including the YOLOv5 model and Flask server, can be containerized using Docker and deployed to cloud environments such as AWS EC2, Google Cloud Run, or Azure App Services. This enables the system to scale horizontally and handle larger datasets or multiple concurrent users without a drop in performance.

Security can also be integrated in future iterations using JWT (JSON Web Tokens) or OAuth2, to support authenticated user access and usage tracking if required by institutional or governmental clients.

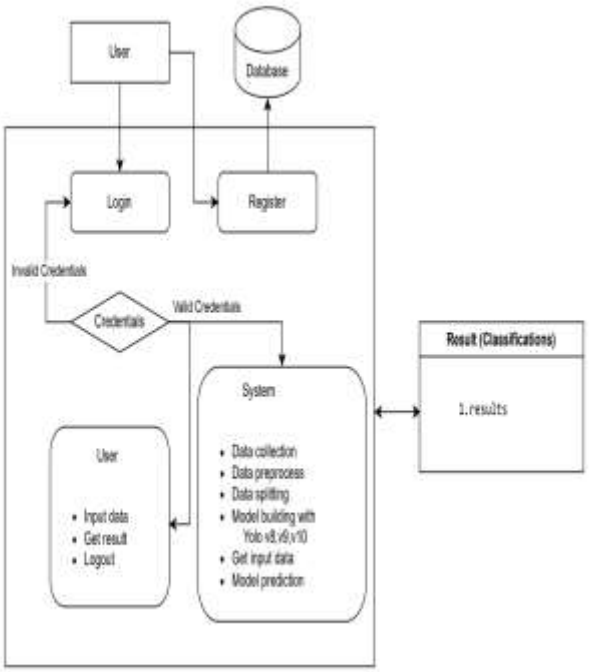

Figure 1. Architecture

V. DATA FLOW AND PROCESS

The Tree Enumeration Using Image Analytics system follows a streamlined, modular data flow that integrates frontend interaction, backend processing, and deep learning inference. The entire process begins with the user uploading an aerial image—typically captured via a drone or satellite—through the web interface. This image serves as the input for the detection system.

Once the image is submitted, the frontend, developed using HTML, CSS, and JavaScript, collects the file and sends it to the backend server via a RESTful API. At this point, Flask, the backend web framework, receives the image and temporarily stores it on the server for processing. This separation of client and server interaction ensures that large image files can be handled efficiently without overloading the user's device.

The backend then invokes a Python-based detection module, where the YOLOv5 model is loaded with pre-trained weights. Before inference, the image undergoes preprocessing using OpenCV, which ensures it is resized, formatted, and optimized for model compatibility. The processed image is passed through YOLOv5, which analyzes it and outputs detection results in the form of bounding box coordinates, confidence scores, and class labels.

- **Image Acquisition and Transmission:** The process begins when a user uploads an aerial image—usually captured via drone or satellite—through a simple web-based interface built using HTML, CSS, and JavaScript. Once the image is selected, it is transmitted to the backend using a RESTful API. This decoupling between frontend and

backend ensures that the system can handle large image files without affecting user experience.

- **Backend Processing and Detection:** Upon receiving the image, the Flask-based Python server temporarily stores it for analysis. The image is then preprocessed using OpenCV to standardize its dimensions and format. The preprocessed image is passed through a YOLOv5 deep learning model, which identifies and counts trees by drawing bounding boxes around each one. Detection outputs include object coordinates, labels, and confidence scores, which are essential for accurate enumeration.
- **Result Rendering and Display:** Once detection is complete, OpenCV is again used to overlay the results on the original image, highlighting each tree detected. The total count and annotated image are then sent back to the frontend. JavaScript dynamically updates the webpage to display the processed image and the final tree count, giving users a visual and numerical understanding of tree distribution.

After detection, the system uses OpenCV again to draw bounding boxes around each detected tree and count the total number of trees. This output image, along with the numeric count, is then sent back to the frontend. JavaScript on the client side dynamically renders the annotated image and displays the count to the user.

This complete data pipeline—from image upload to result visualization—is designed for low latency and high accuracy. It ensures that users receive near-instant feedback, enabling efficient tree enumeration from vast aerial datasets.

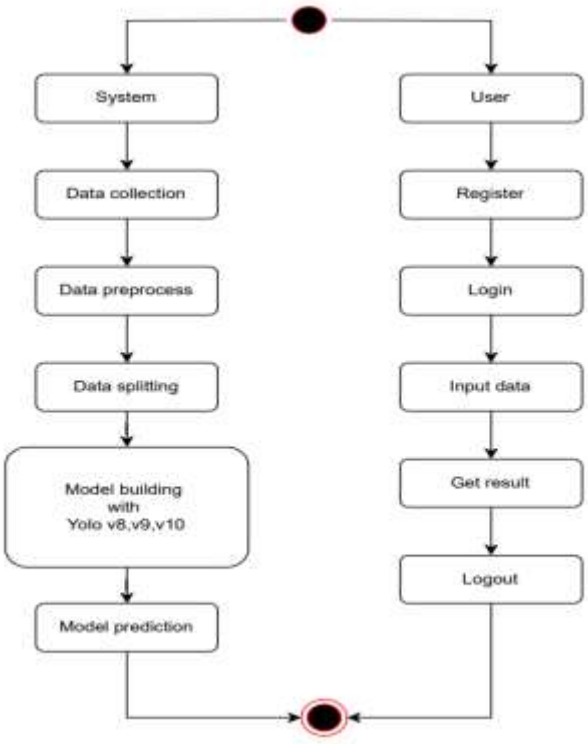

Figure 2 - Data Fetching Architecture

VI. SECURITY CONSIDERATIONS

While the primary focus of this project is on accurate tree enumeration using image analytics, data security and user privacy are also essential considerations—especially when dealing with potentially sensitive geospatial imagery. Though the system was not designed for commercial-scale deployment, best practices were still followed to ensure basic security compliance.

The first level of security is maintained during image upload and transmission. Images submitted by users are sent over HTTP requests, and in future deployments, this should be upgraded to HTTPS to ensure end-to-end encryption. This prevents interception or tampering of the data while in transit between the frontend and backend systems.

On the server side, image files are only stored temporarily and are processed within a controlled environment. No permanent database is used to retain user-submitted images, which helps minimize the risk of unauthorized access or data leaks. Implementing temporary storage limits and auto-deletion after processing further enhances safety. Below are the detailed security considerations:

- **Secure Data Transmission and Temporary Storage:** The system handles image uploads through HTTP requests from the frontend to the backend. To improve security in future deployments, HTTPS should be implemented to ensure encrypted data transmission. Currently, images are stored temporarily on the server only for the duration of processing. Once detection is complete, these images can be auto-deleted, reducing the risk of data leakage or unauthorized access.
- **Input Validation and Attack Prevention:** To protect the backend from malicious inputs, the system enforces strict file validation. Only image files of specific formats (such as .jpg or .png) are accepted. Flask's request handling mechanisms, along with Python's built-in exception handling, help sanitize file names and prevent code injection or file traversal attacks. This ensures that the backend remains stable and resistant to basic exploit attempts.
- **Scalability and Access Control for Future Use:** Although the current prototype does not require user authentication, the architecture allows for scalable security features to be added. If expanded to support multiple users or cloud deployment, secure login mechanisms using JWT (JSON Web Tokens) or OAuth can be integrated. This would help ensure that only authorized users can upload and access image data, strengthening access control and audit capabilities.

In terms of backend logic, the system is safeguarded against common vulnerabilities such as code injection and malformed

file uploads by validating all incoming data. Flask's built-in request handling and Python's exception management are used to sanitize file names and restrict file types to known, safe formats such as JPG and PNG.

While user authentication is not currently implemented—as the application is single-user and prototype-based—it can be added using token-based systems such as JWT if scaled further. This would allow only authorized users to upload and process images, adding an extra layer of access control.

Overall, the system architecture adopts a "process-and-forget" model, where data is not stored long-term and the focus remains on secure, session-based operations. This lightweight but mindful approach ensures that even in a basic implementation, core security principles are respected.

VII. TESTING & DEPLOYMENT

Ensuring the reliability and functionality of the tree enumeration system required a structured approach to both testing and deployment. Given the real-time nature of image-based detection, it was essential to validate each stage of the process—from image upload to final result visualization—to confirm accuracy, robustness, and responsiveness .

VII.1 Testing Phase

Testing was conducted in two main areas: frontend functionality and backend inference accuracy. On the frontend, different image formats, resolutions, and sizes were tested to ensure that the upload mechanism functioned without crashing or losing data. JavaScript console logs and manual test cases were used to identify and fix interface-related bugs, particularly in rendering results and handling slow network conditions.

Backend testing focused heavily on the performance of the YOLOv5 detection model. Multiple drone-captured images with varying densities of tree cover were passed through the model to evaluate its precision in detecting and counting trees. During this phase, OpenCV overlays were cross-verified against the expected number of trees in ground-truthed images. Edge cases such as overlapping trees, low-resolution inputs, and poor lighting conditions were also tested to ensure the model's resilience.

- **Unit Testing of Model Components:** Initial testing was carried out to verify the functionality of the YOLOv5 detection model independently. Sample drone images were processed in isolation to confirm that the model loaded correctly, predicted bounding boxes, and output the expected number of tree detections.

- **Input Image Validation:** Various types of test images—differing in resolution, lighting, and background complexity—were uploaded through the frontend to check if the system could handle them gracefully. Special attention was given to edge cases like blurry images, images with overlapping trees, or low contrast.

- **Backend API Testing:** Flask-based API routes were tested using Postman to simulate image upload requests. The goal was to ensure the server could accept files, trigger model inference, and return results without crashing or timing out. API responses were monitored for latency, error handling, and result consistency.
- **Functional Frontend Testing:** The frontend was tested to ensure a smooth user experience. Tests included verifying that file inputs worked, loading indicators were shown during processing, and the processed image with bounding boxes was displayed correctly. Browser console logs helped identify minor bugs in asynchronous behavior.
- **Model Accuracy Evaluation:** To validate model performance, outputs were visually compared against manually counted trees in sample images. This helped measure the model's detection accuracy and identify under-detection or over-detection scenarios.
- **Integration Testing:** Finally, end-to-end testing was done to simulate the complete flow—from image upload, model processing, to result display. This ensured all components worked together without breaking the pipeline.

VII.II Deployment Phase

The final application was deployed in a local server environment for demonstration purposes. Flask was used to serve the model and handle API endpoints, while the frontend was hosted on a local browser setup. Though this was not a cloud-based deployment, the architecture allows for smooth migration to cloud services like AWS or Google Cloud in the future. In such cases, containerization tools like Docker could be used to encapsulate the application for better scalability and maintainability.

Overall, testing and deployment were iterative and informed by real-world usage scenarios, resulting in a system that is not only functional but also adaptable for future expansion.

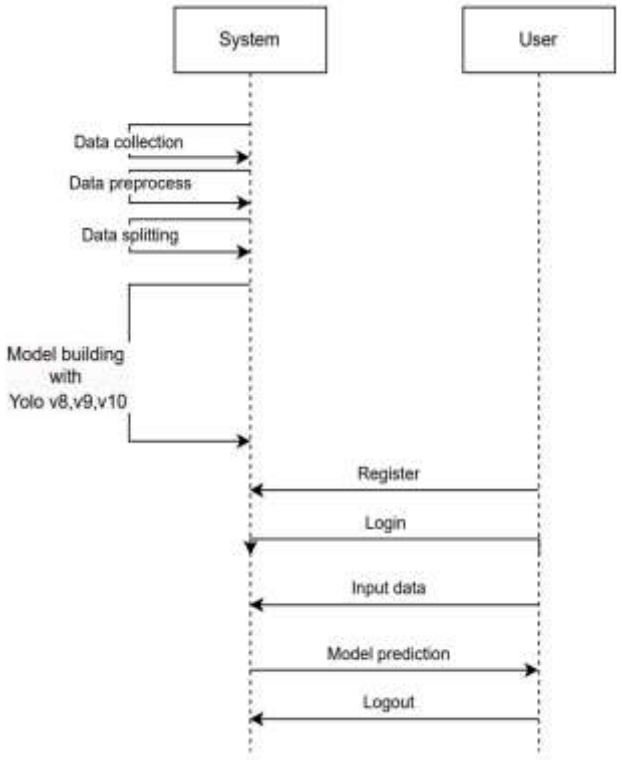

Figure 3 - Deployment diagram

The deployment of the tree enumeration system was carried out in a staged manner, starting with local hosting and keeping future scalability in mind. Below are the detailed steps involved in the deployment process:

- **Environment Setup:** The system was initially deployed in a local development environment. This included setting up Python and Flask for the backend, installing required dependencies using pip, and ensuring compatibility with libraries such as OpenCV and PyTorch (for YOLOv5). A virtual environment was created to isolate dependencies and prevent version conflicts.
- **Frontend Hosting:** The frontend was developed using HTML, CSS, and JavaScript, and was hosted using a local HTTP server. The web interface was linked to Flask's REST API to facilitate image upload and result retrieval. Form submissions were handled asynchronously to ensure smooth user interaction.
- **REST API Configuration:** Flask routes were created for handling POST requests from the frontend. This included secure endpoints for image upload, triggering model inference, and returning results. Proper error handling was implemented to catch exceptions during image processing and inference.
- **Testing in Localhost:** The system was run and tested on localhost to simulate user interactions. Tools like Postman and browser-based testing were used to validate the flow of data and the responsiveness of the application. Logs were monitored to detect any bottlenecks or performance issues.
- **Optional Future-Ready Deployment:** While the system is currently deployed on a local server, it is designed to be cloud-ready. Containerization using Docker is planned for the future, allowing the app to be easily deployed on cloud platforms such as AWS EC2, Google Cloud Run, or Heroku. This will also enable scalability, multi-user access, and persistent storage if needed.
- **Security and Maintenance:** Temporary storage management was configured to ensure that uploaded images were deleted after processing. The system's modular design also allows easy updates to the model or frontend interface without affecting the entire application.

VIII. CHALLENGES & SOLUTIONS

During the development of the tree enumeration system, our team encountered a variety of challenges, ranging from technical limitations to model performance and even usability considerations. Each obstacle required careful troubleshooting, research, and collaboration to resolve. Below are some of the key challenges we faced and the solutions we implemented:

1. Adapting YOLOv5 to Aerial Tree Detection

One of the first major challenges we faced was tuning the YOLOv5 object detection model to work effectively with aerial images of trees. Unlike common use cases of YOLO, such as detecting vehicles or humans, trees captured from above often appeared as irregular shapes with variable shadows and textures. This made detection less accurate in the beginning.

Solution:

To overcome this, we trained the model using a diverse dataset of aerial forest and tree canopy images. We supplemented this with data augmentation techniques such as random rotations, flips, and brightness adjustments to improve model generalization. We also fine-tuned YOLO's confidence threshold to reduce false positives and enhance detection accuracy.

2. Performance Bottlenecks in Image Processing

Another issue arose in the backend while processing high-resolution drone images. These large images caused slow inference times, especially when multiple detections were involved. It also put strain on memory usage, affecting the Flask server's responsiveness.

Solution:

We optimized the image processing pipeline by resizing input images before inference and using efficient OpenCV methods for overlaying bounding boxes. Flask's multithreading capabilities were utilized to handle concurrent requests, and

temporary files were cleared immediately after processing to reduce memory load.

3. Frontend Integration and Image Feedback Loop

Connecting the frontend with the backend was a challenge, particularly when displaying the result image (with tree detections) back to the user. File handling and asynchronous communication often resulted in broken responses or blank result areas.

Solution:

We used JavaScript's fetch API along with FormData objects to send images via POST requests and receive processed images as blobs. On the Flask side, we ensured the response returned the image as a downloadable stream. This helped maintain a smooth data flow between the frontend and backend. We also added simple loading animations and error messages to inform the user in case of issues.

4. Dealing with Low Accuracy in Dense Vegetation

In areas with dense tree cover, the model sometimes miscounted trees—either grouping multiple trees together or failing to detect some entirely due to shadow overlap or similar texture in the background.

Solution:

We manually analyzed such cases and experimented with different image contrast settings before feeding them into the model. In future versions, we plan to use multi-scale detection techniques or ensemble models to improve performance in densely forested regions.

5. Lack of Real-time Testing Opportunities

Since the model was designed to work with drone images, we couldn't frequently test it with real-time drone footage due to limited access to drones and flight permissions.

Solution:

We gathered publicly available aerial datasets and simulated drone input by stitching images or creating test batches. While this wasn't a perfect replacement for real-time testing, it helped us iterate quickly during development and validate the model in various conditions.

6. Deployment Constraints

Due to resource limitations, we couldn't host the system on a cloud server, which would have allowed remote usage and better scalability. Local deployment restricted testing to a small environment and limited user access.

Solution:

We structured the codebase and API routes to be easily portable to a cloud platform in the future. With minor changes and containerization (e.g., using Docker), the entire application can be migrated to services like AWS EC2 or Google Cloud.

These challenges were not just roadblocks—they became part of the learning journey that helped us improve our technical understanding, collaboration skills, and problem-solving mindset. By facing these hurdles head-on, we were able to build a more stable and effective tree enumeration system.

IX. FUTURE ENHANCEMENTS

While the current version of the tree enumeration system successfully detects and counts trees from aerial images, there is plenty of room for growth and innovation. Several future enhancements have been identified that could significantly improve both the performance and usability of the system. One of the most promising upgrades is integrating GPS and geotagging features. By linking the image data with location metadata, we could provide users with not just the number of trees, but also their exact locations on a map. This would be incredibly valuable for forest management, environmental audits, and afforestation tracking.

We also aim to deploy the application to the cloud, making it accessible as a web-based service. Currently, the system runs locally, limiting its use to a single environment. By moving to platforms like AWS or Google Cloud, users from different locations could upload images and receive results without needing to install anything. Another key enhancement is to improve model performance on dense forests. In areas where tree canopies overlap, YOLOv5 sometimes struggles to detect individual trees. Exploring more advanced models like YOLOv8 or even incorporating instance segmentation methods (e.g., Mask R-CNN) could help solve this issue.

Lastly, we'd like to build a feedback loop where users can manually correct or confirm detections, which the model can learn from over time. This would enable continuous improvement and adapt the system to new terrains and tree species.

X. CONCLUSION

The development of the Tree Enumeration Using Image Analytics system has been a rewarding journey filled with both technical challenges and valuable learning experiences. The project successfully demonstrates how modern image processing techniques and deep learning models like YOLOv5 can be applied to solve real-world environmental problems, specifically in the field of forestry and green cover assessment. By automating the detection and counting of trees from aerial images, the system offers a scalable and efficient alternative to traditional manual surveys. It reduces the time, effort, and human error typically involved in tree census

activities, making it a practical solution for government bodies, researchers, and environmental organizations alike. The integration of a simple frontend interface with a Flask-based backend ensures that users can interact with the model easily, upload images, and receive visual results with marked detections.

Throughout the process, we encountered numerous obstacles—from handling diverse image inputs to optimizing model accuracy—but each challenge contributed to a deeper understanding of model tuning, data flow, and system integration. Although the project was deployed locally due to resource limitations, the architecture is designed with future scalability in mind. Looking ahead, there are exciting opportunities to enhance the system with features like GPS integration, cloud deployment, and feedback-based learning. These improvements could help position the tool as a valuable asset in environmental monitoring and smart city planning.

In essence, this project represents a small but meaningful step toward harnessing AI for sustainable development and ecological awareness. It shows how technology can support data-driven decision-making in environmental conservation. With further development, this system has the potential to be adopted on a larger scale and become an essential tool in digital forestry initiatives.

XI. REFERENCES

[1] A. Landstrom and M. J. Thurley, ‘‘Morphology-based crack detection for steel slabs,’’ IEEE J. Sel. Topics Signal Process., vol. 6, no. 7, pp. 866–875, Nov. 2012.

[2] K. Song and Y. Yan, ‘‘A noise robust method based on completed local binary patterns for hot-rolled steel strip surface defects,’’ Appl. Surf. Sci., vol. 285, pp. 858–864, Nov. 2013, doi: 10.1016/j.apsusc.2013.09.002.

[3] Y. J. Jeon, D. Choi, S. J. Lee, J. P. Yun, and S. W. Kim, ‘‘Steel-surface defect detection using a switching-lighting scheme,’’ Appl. Opt., vol. 55, no. 1, pp. 47–57, 2016, doi: 10.1364/AO.55.000047.

[4] R. Girshick, J. Donahue, T. Darrell, U. Berkeley, and J. Malik, ‘‘R-CNN: Region-based convolutional neural networks,’’ in Proc. Comput. Vis. Pattern Recognit., Jun. 2014, pp. 2–9. 148824 VOLUME 12, 2024 T. Zhang et al.: GDM-YOLO: A Model for Steel Surface Defect Detection Based on YOLOv8s

[5] R. Girshick, ‘‘Fast R-CNN,’’ in Proc. IEEE Int. Conf. Comput. Vis. (ICCV), Dec. 2015, pp. 1440–1448.

[6] K. He, X. Zhang, S. Ren, and J. Sun, ‘‘Spatial pyramid pooling in deep convolutional networks for visual recognition,’’ IEEE Trans. Pattern Anal. Mach. Intell., vol. 37, no. 9, pp. 1904–1916, Sep. 2015, doi: 10.1109/TPAMI.2015.2389824.

[7] S. Ren, K. He, R. Girshick, and J. Sun, ‘‘Faster R-CNN: Towards real-time object detection with region proposal networks,’’ IEEE Trans. Pattern Anal. Mach. Intell., vol. 39, no. 6, pp. 1137–1149, Jun. 2017.

[8] Y. Xu, D. Li, Q. Xie, Q. Wu, and J. Wang, ‘‘Automatic defect detection and segmentation of tunnel surface using modified mask R-CNN,’’ Measurement, vol. 178, Jun. 2021, Art. no. 109316.

[9] M. Chen, L. Yu, C. Zhi, R. Sun, S. Zhu, Z. Gao, Z. Ke, M. Zhu, and Y. Zhang, ‘‘Improved faster R-CNN for fabric defect detection based on Gabor filter with genetic algorithm optimization,’’ Comput. Ind., vol. 134, Jan. 2022, Art. no. 103551.

[10] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, ‘‘You only look once: Unified, real-time object detection,’’ in Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), Jun. 2016, pp. 779–788.

[11] C.-Y. Wang, A. Bochkovskiy, and H.-Y.-M. Liao, ‘‘YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors,’’ in Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR), Jun., vol. 2023, pp. 7464–7475.

[12] J. Redmon and A. Farhadi, ‘‘YOLOv3: An incremental improvement,’’ 2018, arXiv:1804.02767.

[13] Z. Ge, S. Liu, F. Wang, Z. Li, and J. Sun, ‘‘YOLOX: Exceeding YOLO series in 2021,’’ 2021, arXiv:2107.08430.

[14] M. Ma and H. Pang, ‘‘SP-YOLOv8s: An improved YOLOv8s model for remote sensing image tiny object detection,’’ Appl. Sci., vol. 13, no. 14, p. 8161, Jul. 2023, doi: 10.3390/app13148161.

[15] H. Nie, H. Pang, M. Ma, and R. Zheng, ‘‘A lightweight remote sensing small target image detection algorithm based on improved YOLOv8,’’ Sensors, vol. 24, no. 9, p. 2952, May 2024.