YOOBEE COLLEGE OF CREATIVE INNOVATION

Comparative Analysis and Integration of YOLOv7 and YOLOv8 in a Streamlit Application

A Capstone Project Report Submitted in Partial Fulfillment
of the Requirements for the Degree of
Bachelor of Software Engineering

Irina Getman

Supervisory team: Dr. Mohammad Norouzifard

Faculty of Technology
Yoobee College of Creative Innovation
Auckland, New Zealand
November 23, 2023

Contents

Introduction

In the dynamic field of computer vision, object detection stands as a critical component, driving numerous practical applications. At the forefront of this domain is the YOLO (You Only Look Once) algorithm, renowned for its efficiency and accuracy. This document presents an in-depth overview of an application developed to harness the capabilities of the latest iterations in the YOLO series - YOLOv7 and YOLOv8. The application is designed not just as a tool for object detection but as a comprehensive platform for users to interact with, evaluate, and understand these advanced models in various scenarios.

Our application marks a significant step forward in making object detection technology more accessible and user-friendly. Moving beyond niche use cases like vehicle detection under challenging lighting conditions, it embraces a broader spectrum of object detection. This approach empowers users to select and detect objects of their choice, demonstrating the versatility and robustness of the YOLO models. The app's interface focuses on providing clear metric displays and visual comparison options, enhancing user engagement and understanding.

This document details the objectives, development process, deployment strategy, and time management aspects of the application. It delves into the hurdles encountered during development, outlining the solutions implemented to overcome these challenges. A thorough analysis of the app's performance, based on user interactions and feedback, is also presented. The aim is to provide a holistic view of the application's development journey, from conception to deployment, and its impact in the field of computer vision and object detection.

1 Background

In the initial phase of our journey into computer vision and object detection, the focus was concentrated on developing an application tailored for vehicle detection using the YOLOv8 model. This project, which formed the cornerstone of my work in the first semester (301), was geared towards harnessing the power of advanced object detection algorithms in practical, real-world scenarios. The application developed during this phase featured robust capabilities for image, video, and webcam inferences, providing a versatile platform for vehicle detection.

Building upon the foundation laid in the first semester, our capstone project represents a significant evolution and expansion of the original application. The primary aim was to extend the utility and scope of the application beyond vehicle detection, transforming it into a more comprehensive tool for general object detection. This shift in focus was driven by a desire to make the application more versatile and user-friendly, enabling it to cater to a broader spectrum of object detection scenarios.

To enhance the application's capabilities, several key improvements and new features were introduced:

- Transitioning from a vehicle-specific detection system to a more inclusive object detection application, allowing users to choose from a variety of objects for detection.
- A redesigned and more intuitive user interface was implemented, making the app more accessible and user-friendly. This included streamlined navigation and a clearer presentation of detection results.
- Our application has transformed from a single-page platform into a multi-page system, offering advanced features exclusively for registered users.
- The application now includes the capability to compare the performance of different YOLO models, specifically YOLOv7 and YOLOv8. This feature provides users with valuable insights into the strengths and weaknesses of each model.
- Users have the ability to personalize detection settings, compare model inference times, access historical detection results, and sift through archived data for deeper insights.

The enhancements made to the original app underscore a commitment to advancing the field of object detection using cutting-edge technology. By broadening the scope of the application and introducing new features, the project not only serves as a testament to the versatility of YOLO models but also paves the way for their practical application in diverse domains. This capstone project, therefore, stands as a crucial step in the journey of exploring and leveraging the capabilities of AI and computer vision in real-world scenarios.

2 Advanced Streamlit Features and Integrations

In continuation of our journey with Streamlit, during the course of 302, we delved deeper into its more sophisticated capabilities, focusing on advanced features and integrations that enhance the functionality and user experience of the applications developed.

This exploration was not just theoretical; it involved practical implementation, allowing me to gain a comprehensive understanding of Streamlit's capabilities beyond the basics.

Firebase Integration for Enhanced Functionality: A significant aspect of this exploration was the integration of Firebase, a powerful platform by Google, known for its robust backend capabilities. This integration was multifaceted, encompassing several key areas:

- **Authentication Process:** By integrating Firebase, we implemented a secure authentication process in the Streamlit application. This allowed for a reliable and efficient way to manage user access and maintain data security.
- **Data Storage and Database Management:** Firebase also served as an excellent solution for data storage and database management within the Streamlit application. Its flexible and scalable database options provided a seamless way to store, retrieve, and manage application data.

Session State Management: A key feature we learned and frequently utilized was Streamlit's session state management. This powerful tool allowed for a more dynamic and responsive application, maintaining user interactions and data state across sessions. The practical application of this feature was instrumental in creating a cohesive and user-friendly interface.

User Experience Enhancements: To enhance the user experience, we implemented several UI elements:

- **Progress Bar:** During the detection process, a progress bar was integrated using Streamlit's built-in functionality. This provided users with a visual cue of the process, making the wait more informative and engaging.
- **Notification Widgets:** The application also utilized Streamlit's notification widgets like `st.warning`, `st.success`, and `st.info`. These were key in gracefully handling errors and displaying useful information, thereby improving the overall user interaction with the application.

As shown in Figure 1, the application features an intuitive progress bar and a variety of notification widgets for enhanced user interaction.

2.1 Advanced Visualization with Plotly

Integration of Plotly for Image Display: A significant advancement in our project was the integration of `st.plotly_chart` for displaying detected images. This feature not only enhanced the visual aspect of the application but also provided practical benefits:

- **Interactive Image Features:** Users could interact with the images by dragging and zooming, offering a more detailed examination of the detection results.
- **Ease of Image Download:** The functionality extended to include a one-click download option for detected images, greatly simplifying the process for users.

Figure 2 demonstrates the advanced visualization capabilities in our application, utilizing st.plotly_chart for interactive image display and analysis.

As we conclude this section on Streamlit features, we have laid a foundational understanding of the advanced capabilities and integrations that Streamlit offers, including Firebase integration, session state management, user experience enhancements, and sophisticated data visualization using `st.plotly_chart`. With this groundwork established, we now transition into a detailed week-by-week implementation process. In the following sections, we will delve into each of these features in greater depth, exploring their practical applications and the specific steps involved in their implementation. This approach will not only provide a comprehensive insight into the technical aspects but also demonstrate the real-world applicability of these features in enhancing the functionality and user experience of Streamlit applications.

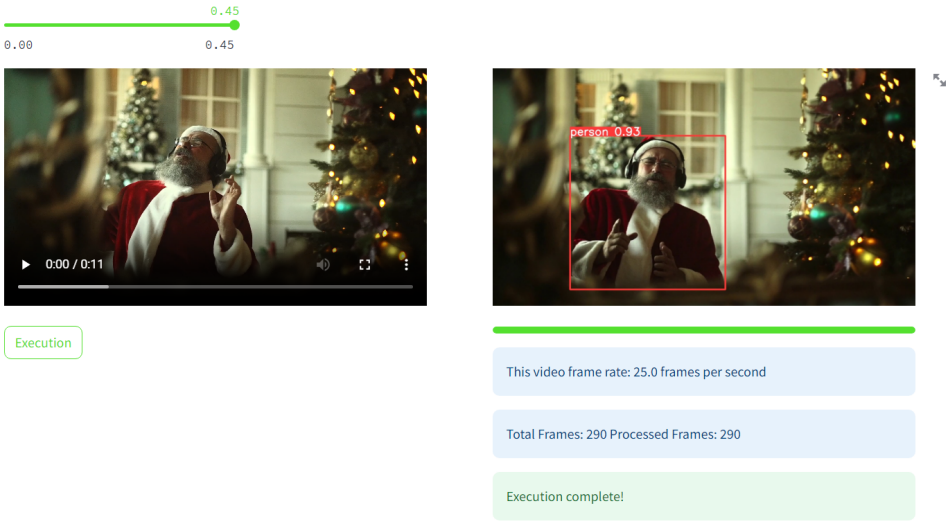


Figure 1: Streamlit Application's Progress Bar and Notification Widgets

Figure 2: Advanced Visualization using st.plotly_chart in Streamlit

3 Project Management

3.1 Week-by-Week Breakdown

The development of the project was meticulously planned over several weeks, each with its distinct set of goals and challenges. The following table provides an overview of the weekly breakdown:

Week	**Activities**
Week 1	Initial Setup and Challenges
Week 2-3	Implementation of YOLOv8 and Troubleshooting
Week 4	Decision to Focus on YOLOv7 and YOLOv8
Week 5-6	Integration of YOLOv7 and Overcoming Technical Challenges
Week 7	Finalization and Testing

Table 1: Week-by-Week Breakdown of Project Activities

3.2 Time Management and Adjustments to Project Plan

Throughout the project, time management was a critical aspect. Adjustments were made to the original project plan to accommodate unexpected challenges and ensure the successful integration of new features.

3.3 Tools and Technologies Used

The development of the application was supported by a diverse array of tools and technologies, each playing a crucial role in different aspects of the project. This section provides an in-depth overview of these components.

3.3.1 YOLO Models

The application incorporated various YOLO models to offer a comprehensive object detection experience:

- **YOLOv8 Models:**
  - *YOLOv8x:* Optimized for high accuracy.
  - *YOLOv8l:* Large model offering a balance between speed and accuracy.
  - *YOLOv8m:* Medium-sized model, suitable for moderate hardware.
  - *YOLOv8n:* Nano model, designed for speed in low-resource environments.
  - *Custom-trained YOLOv8:* Tailored for specific object detection tasks.
- **YOLOv7 Models:**
  - *YOLOv7:* Standard model providing robust detection capabilities.
  - *YOLOv7x:* Extended model for enhanced accuracy.
  - *YOLOv7-E6:* Efficient model for energy-saving scenarios.
  - *YOLOv7-W6:* Wide model for comprehensive object detection.
  - *Custom-trained YOLOv7:* Adapted to specific detection requirements.

3.3.2 Programming Language and Libraries

The primary programming language used was Python, due to its versatility and extensive support for machine learning and computer vision libraries. Key libraries and frameworks included:

- **Python Libraries:**
  - *Matplotlib, Pillow, PyYAML, Requests, SciPy, Tqdm:* For data processing and visualization.
  - *OpenCV-Python:* Essential for image and video processing.
  - *EasyDict, Plotly:* Used for data manipulation and interactive plotting.
  - *Pandas, Seaborn:* For advanced data analysis and statistical plotting.
  - *Numpy, Protobuf, IPython, THOP:* For numerical computations and model evaluation.
  - *Psutil, Moviepy, Openpyxl, Memory Profiler:* For system monitoring, video processing, spreadsheet manipulation, and memory profiling.
- **Deep Learning Frameworks:**
  - *TensorFlow, Torch, Torchvision:* Core frameworks for neural network development and training.
- **Deployment and Authentication:**
  - *Streamlit, Streamlit-extras, Streamlit-authenticator, Streamlit-lottie:* For web app deployment and enhancing user interface features.
  - *Firebase Admin, Python-dotenv:* For database management and environment variable handling.

– *AWS EC2 Deployment:* Used for scalable and secure cloud hosting of the application, ensuring high availability and robust performance.

The combination of these tools and technologies facilitated the development of a robust, efficient, and user-friendly object detection application, capable of leveraging the latest advancements in the field of computer vision.

In the next section, we will delve deeper into the specific challenges encountered during each phase of the project and the strategies employed to overcome them.

4 Technical Implementation

This project embarked on an ambitious journey to integrate the latest advancements in object detection algorithms, particularly YOLOv7 and YOLOv8, into a comprehensive application. The core of this project hinged on successfully incorporating these cutting-edge models while ensuring optimal data handling and an intuitive user interface. The integration process was not without its challenges, necessitating innovative solutions and workarounds to address issues ranging from technical incompatibilities to performance optimization.

At the heart of the application lies the integration of YOLOv7 and YOLOv8 models. This process involved not only embedding these models into the application framework but also ensuring they operated efficiently and accurately. The decision to include multiple versions of YOLOv8 (such as v8x, v8l, v8m, v8n, and custom-trained models) along with various iterations of YOLOv7 (including v7, v7x, v7-e6, v7-w6, and custom-trained v7) added layers of complexity to the integration process.

Data handling was another critical aspect of this project. The shift to Firebase for database management was a strategic decision aimed at improving the application's scalability and reliability. This transition brought its unique set of challenges, especially in terms of user authentication and session state management in Streamlit.

The user interface, a pivotal component of any application, was designed with a focus on user experience. Special attention was given to session state management and interactive elements like avatar handling, which required meticulous planning and execution.

Performance optimization was a continuous endeavor throughout the development process. The application faced several memory allocation challenges, necessitating efficient caching strategies and workflow optimizations. These were instrumental in enhancing the application's stability and responsiveness.

Finally, the project encountered various challenges that required creative solutions and workarounds. From inconsistencies in model name handling to complexities in managing results from different YOLO models, each hurdle was met with a solution-oriented approach. This section of the report delves into these aspects, detailing the strategies employed to overcome these challenges and ensure the successful implementation of the application.

4.1 Handling of Data and User Interface

4.1.1 Transition to Firebase

Initially, the project was set to leverage **Deta Space** as the preferred cloud solution for data management. Deta Space presented an attractive choice due to its capabilities for hosting and managing data in a cloud-based environment. However, an unforeseen challenge arose when the Deta Space platform encountered downtime, as reflected on their website. This unexpected downtime significantly obstructed the project's progress, necessitating a prompt resolution to maintain the project timeline.

After a meticulous evaluation of available alternatives, the decision was made to transition to **Firebase**. Firebase offered a robust and reliable cloud solution capable of accommodating the project's data management requirements without the downtime issues encountered with Deta Space. This transition entailed configuring the Firebase Admin SDK to interact with Firebase services, as elucidated in the previous section.

The switch to Firebase turned out to be a wise choice. It provided a stable, reliable, and feature-rich platform that facilitated not only user authentication and data storage but also other essential functionalities. This transition not only rectified the downtime issue but also established a robust foundation for the subsequent developmental phases of the project.

After switching to Firebase , we planned to use a `pyrebase` as a wrapper for Python language while working with Firebase. However, due to incompatibilities with the existing modules in my project, we had to find some workarounds, especially when logging registered users into the app.

In the script, a configuration file named `config.json` stores and retrieves Firebase configuration details. Upon reading these details into the script, the Firebase Admin SDK is initialized using a service account key alongside the database URL obtained from the configuration. This setup lays the groundwork for the subsequent user authentication and registration processes. The script defines two primary functions, login_user and `authenticate`, to handle user interactions. The narrative further explains the functionalities and error-handling mechanisms incorporated in these functions. Refer to figures 3 and 4 for a visual representation of the implemented solution.

```
def login_user(email, password):
    # Load firebaseConfig from config.json
    with open('config.json') as config_file:
        config_data = json.load(config_file)
    firebaseConfig = config_data['firebaseConfig']

    url = f"https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key={firebaseConfig['api
    ↪  Key']}"
    data = {
        "email": email,
        "password": password,
        "returnSecureToken": True
    }
    response = requests.post(url, data=data)
    if response.ok:
        user_data = response.json()
        handle_login_bt(user_data)
        st.write(st.session_state.user)
    else:
        st.warning('Login failed. Please check your credentials and try again.')
```

Figure 3: Code snippet for user login function.

4.1.2 Session State in Streamlit

Streamlit's session state allows data to be shared across reruns without using global variables. It is extremely useful for maintaining the state in interactive Streamlit applications, for example, during user login sessions.

In the provided code snippet as shown in Figure 5, a function named `display_sidebar` is defined, which manages user authentication and actions within the sidebar of a Streamlit application. Initially, it checks the `authenticated` attribute in the session state to determine if a user is logged in. If not, a "Login/Sign Up" button is displayed. Once clicked, the page is redirected to the authentication page using Streamlit's `rerun` function.

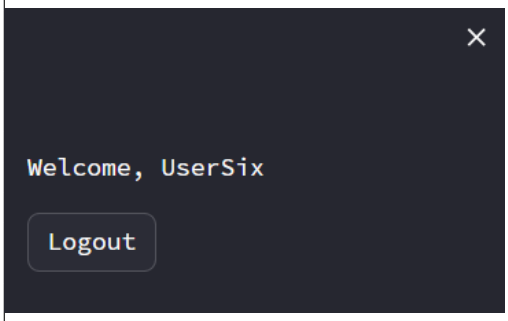
On successful authentication, the sidebar displays a welcoming message along with the authenticated user's username. Additionally, a "Logout" button is provided. When this button is clicked, the user information is removed from the session state, the `authenticated` state is set to `False`, a success message is shown to confirm logout, and after a brief pause, the page is redirected back to the main page using the `rerun` function again. This demonstrates how the session state in Streamlit can be effectively utilized to manage user authentication and provide dynamic interface updates based on user interactions.

```python
def authenticate():
    col1, col2 = st.columns(2)

    with col1:
        st.subheader(':green[Login]')
        login_email = st.text_input(':blue[email]', placeholder='enter your email', key='login_email')
        login_password = st.text_input(':blue[password]', placeholder='enter your password',
        ↪  type='password', key='login_password')

        # def handle_login_bt()
        if st.button('Login'):
            login_user(login_email, login_password)

    with col2:
        st.subheader(':green[Sign Up]')
        email = st.text_input(':blue[email]', placeholder='enter your email', key='signup_email')
        username = st.text_input(':blue[username]', placeholder='enter your username',
        ↪  key='signup_username')
        password = st.text_input(':blue[password]', placeholder='enter your password', type='password',
        ↪  key='signup_password', help='Password must be at least 6 characters long.')

        if st.button('Create an account'):
            email_exists = verify_user(email=email)
            username_exists = verify_user(username=username)

            if email_exists:
                st.error('Email is already in use.')
                return
            if username_exists:
                st.error('Username is already taken.')
                return

            user = auth.create_user(email=email, password=password)

            # If user creation was successful, store the username in Firebase Realtime Database
            if user:
                user_id = user.uid
                ref = db.reference(f'/users/{user_id}')
                ref.set({'username': username})
                # Store user info in session_state
                st.session_state.user = {"username": username, "uid": user.uid}
                st.success('Account created successfully! Please log in now.')
                st.balloons()
                time.sleep(3)
                st.session_state.page = 'main'  # Redirect to main page
                st.rerun()
            else:
                st.error('Account creation failed. Please try again.')
```

Figure 4: Code snippet for authenticate function.

```python
# main.py

def display_sidebar():
    if not st.session_state.get('authenticated', False):
        if st.sidebar.button("Login/Sign Up",
        ↪  key="login_signup_button"):
            st.session_state.page = 'authentication'
            st.rerun()
    else:
        user_id = st.session_state.user["uid"]
        st.sidebar.write(f'Welcome,
        ↪  {st.session_state.user["username"]}')

        #--------Logout button--------------
        if st.sidebar.button("Logout", key="logout_button"):
            st.session_state.pop('user', None)  # Remove user
            ↪  info from session_state
            st.session_state.authenticated = False  # Set
            ↪  authenticated state to False
            st.sidebar.success('Logged out successfully.')
            time.sleep(2)
            st.session_state.page = 'main'  # Redirect to main
            ↪  page
            st.rerun()
```

(a) Code



(b) Output

Figure 5: Streamlit code on the left demonstrating session state usage to maintain a counter across reruns, with the corresponding output on the right.

4.1.3 Storing Avatars

The task of storing avatars involved configuring the Firebase Admin SDK to interact with Firebase services. A function named `store_avatar()` was created to handle avatar uploads. This function generates a unique filename for each avatar, uploads the file to Firebase storage, and updates the user's record in Firebase Realtime Database with the URL of the uploaded avatar. Handling different image formats and ensuring that the files were stored correctly in the Firebase storage bucket were crucial steps in this process. Several challenges, such as correctly configuring Firebase and managing different file formats, were encountered, but solutions were found through debugging and consulting documentation.

As shown in Figure 6, the `store_avatar()` function handles the process of storing a user-uploaded avatar in Firebase and updating the user's record in the database. The function first configures a connection to Firebase storage and then determines the file extension based on the mime type of the uploaded file. A unique filename is generated based on the user's ID and the file extension. The file object's position is reset to the beginning of the stream to ensure correct uploading. The file is then uploaded to Firebase, made publicly accessible, and the URL of the uploaded avatar is stored in the user's record in the Firebase Realtime Database.

```python
def store_avatar(user_id, file):

    bucket = storage.bucket('capstone-c23c5.appspot.com')

    # Determine the file extension based on the mime type
    mime_type = file.type  # assuming file.type returns the mime type
    extension = ""
    if mime_type == "image/png":
        extension = ".png"
    elif mime_type == "image/jpeg":
        extension = ".jpg"

    if not extension:
        raise ValueError("Unsupported file type")

    # Create a unique file name based on user_id and file extension
    blob = bucket.blob(f'avatars/{user_id}{extension}')

    # Reset the file object's position to the beginning of the stream
    file.seek(0)

    # Upload the file
    blob.upload_from_file(file, content_type=mime_type)

    # Make the file publicly accessible
    blob.make_public()
    avatar_url = blob.public_url

    # Update the user's avatar URL in the Firebase Realtime Database
    ref = db.reference(f'/users/{user_id}/avatar')
    ref.set(avatar_url)

    # Return the public URL to the file
    return avatar_url
```

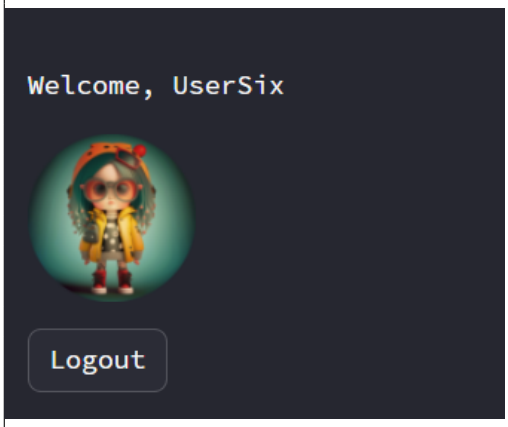
Figure 6: The `store_avatar()` function from the avatar manager script demonstrates the process of storing a user-uploaded avatar in Firebase and updating the user's record in the database.

4.1.4 Rendering Avatars

The subsequent task focused on rendering the stored avatars on the user interface. Initially, a function named `get_avatar_url()` was crafted to fetch the URL of a user's avatar from the Firebase Realtime Database. With the URL obtained, the Streamlit library was employed to design a sidebar where the avatar would be showcased. An additional effort was made to render the avatar in a circular shape to enhance the visual appeal. This was achieved using the Python Imaging Library (PIL), which provided the necessary tools to manipulate the image shape before presenting it on the interface. A smooth process for displaying avatars was set up through these steps, improving the user interface. Initially, avatars were shown twice due to extra code, but this was fixed by removing the code that displayed the original avatar image.

```python
# main.py

def display_sidebar():
    user_id = st.session_state.user["uid"]
    avatar_url = get_avatar_url(user_id)

    # Check if the user has an avatar
    if avatar_url:
        response = requests.get(avatar_url)
        avatar = Image.open(BytesIO(response.content))

        # Create a mask of a filled circle
        mask = Image.new("L", avatar.size)
        draw = ImageDraw.Draw(mask)
        draw.ellipse((0, 0) + avatar.size, fill=255)

        # Apply the mask to the avatar
        rounded_avatar = Image.new("RGBA", avatar.size)
        rounded_avatar.paste(avatar, mask=mask)

        # Display the rounded avatar in Streamlit
        st.sidebar.image(rounded_avatar, width=100)
```

(a) Code



(b) Output

Figure 7: Streamlit code on the left for rendering a rounded avatar, with the corresponding output on the right.

As demonstrated in Figure 7, the code snippet showcases the procedure of rendering a rounded avatar using Streamlit and PIL. The corresponding output is displayed on the right-hand side of the figure.

4.1.5 Error Handling in User Verification Process

In the initial approach to user verification, two separate functions were employed: `verify_user` and `is_password_correct`. The `verify_user` function was designed to check if a user's email was already registered in the database, while `is_password_correct` was supposed to validate the password against certain criteria set by Firebase, such as password must be equal or longer than six characters. To avoid unnecessary errors, we already gave users the hint to provide passwords at least six characters long. However, this approach's granularity and separation of concerns led to a cluttered and less intuitive error-handling process.

The revised approach encapsulates the error handling within a singular function using the `auth` module, simplifying the code and making the error handling process more graceful and user-friendly. In this method, error messages are generated and managed more centrally, allowing for a cleaner and more maintainable code structure.

Below is the code snippet of the new approach to user verification and error handling using the `auth` module:

```
from firebase_admin.auth import EmailAlreadyExistsError
def authenticate():
    ....
   if st.button('Create an account'):

            try:
                user = auth.create_user(email=email, password=password)
            # If user creation was successful, store the username in Firebase Realtime Database
                if user:

                        user_id = user.uid
                        ref = db.reference(f'/users/{user_id}')
                        ref.set({'username': username})
                        # Store user info in session_state
                        st.session_state.user = {"username": username, "uid": user.uid}
                        st.success('Account created successfully! Please log in now.')
                        st.balloons()
                        time.sleep(3)
                        st.session_state.page = 'main'  # Redirect to main page
                        st.rerun()

            except EmailAlreadyExistsError:
                    st.error('The email address is already in use. Please use a different email address.')
            except Exception as e:
                    st.error(f'An error occurred: {e}')
```

Figure 8: Revised User Verification proccess using `auth` Module

This new approach (as shown in Fig.8) provides a unified function `authenticate`, which handles both the verification of user information and the graceful management of errors that may occur during this process. The process is streamlined by consolidating the error handling into a single function and leveraging the `auth` module, and the code becomes significantly easier to follow and maintain.

4.2 Performance Comparison of Different YOLOv8 Models

Before integrating different versions into our app, we tried different ways of efficient visual comparison using only YOLOv8 models. This section documents the development stage of a module focused on comparing the performance of different YOLO models using uploaded images.

The script `comparison.py` is designed to compare the performance of various models on user-uploaded images. This provides insights into how each model detects and identifies objects, helping to determine the most accurate or suitable model for a specific application. In Figure 9, we can observe the detection results of two YOLO models: YOLOv8l and YOLOv8m. This side-by-side comparison visually represents each model's capabilities and performance on the same image.

Main Functionalities

- Users can select up to four models for performance comparison.
- The application provides an interface for users to upload images.
- Each model's inference results are displayed side by side for easy comparison.
- Detected objects are highlighted in the images with their respective probabilities.
- Users can save the results to Firebase for future reference.

Saving to Firebase The function `save_to_firebase` pushes the inference data to Firebase. A unique ID is generated for each inference entry using the push method. Successful data upload is indicated with a success message, while errors are displayed to the user.

Comparing Models The core function `compare_models_function` facilitates the selection of models, uploading images, and comparing model performances. Key functionalities include:

- Model selection through a multi-select option.
- Model confidence can be adjusted with a slider.
- The selected image undergoes inference by each of the chosen models.
- Results are displayed in two formats: visual detections and tabulated results. The tabulated results offer detailed insights like object type, coordinates, and probability.
- For logged-in users, an option to save results to Firebase is provided.

Saved inferences are processed further upon a request from verified users.

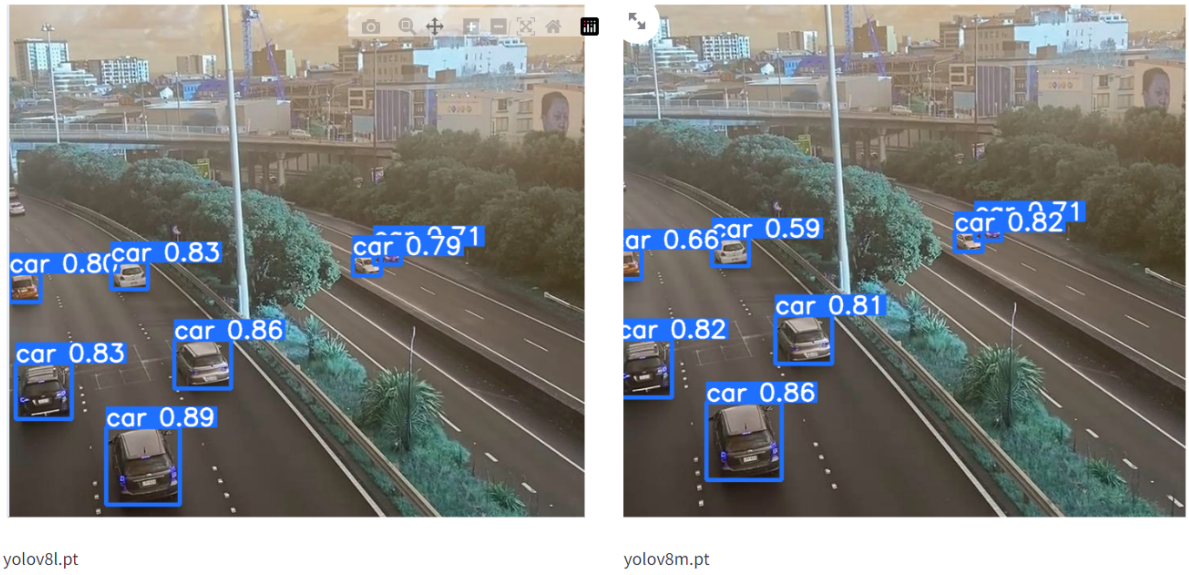


Figure 9: Comparison of YOLOv8l and YOLOv8m Detection Results

4.3 Integration of YOLOv7 and YOLOv8

The integration of YOLOv7 and YOLOv8 models into our application posed several significant challenges, each requiring meticulous attention and innovative problem-solving approaches.

4.3.1 Custom YOLOv7 Wrapper

Integrating YOLOv7 into our pipeline presented a set of distinct challenges, demanding a combination of innovative thinking and meticulous planning. A critical solution was designing and implementing a custom wrapper to enable smooth integration with the YOLOv7 model. A notable challenge was the limited documentation available for integrating YOLOv7 with Streamlit. We drew inspiration from Steven Smiley's Medium article titled *Train & Deploy Yolov7 to Streamlit* (Smiley, 2022). Yet, the intricacy of our application made working directly from the *yolov7* directory, as the article suggests, unfeasible, hence the need for our wrapper.

The creation of a YOLOv7 wrapper served several key purposes:

- The wrapper encapsulates YOLOv7 intricacies, furnishing a sleek and modular interface. A simplistic visualization is the method `read_image_from_path`, allowing image reading without diving into OpenCV specifics.
- Tailored configurations become straightforward, like switching among models or adjusting confidence thresholds. For instance, by using the `detect_and_draw_boxes_from_np` method, different models can be selected for various detection tasks.
- The wrapper's design ensures organized error management. For example, while the core model may raise exceptions during inference, the wrapper manages and logs them gracefully.
- Our architecture is future-proof. Whether adding more models or integrating novel features, enhancements are seamless. One can visualize this by considering the integration of batch processing or video processing without altering existing methods.

To provide a more precise understanding, we've illustrated the wrapper's architecture and its principal features in Figure 10.

This section will dive into the challenges faced during this phase and the solutions taken.

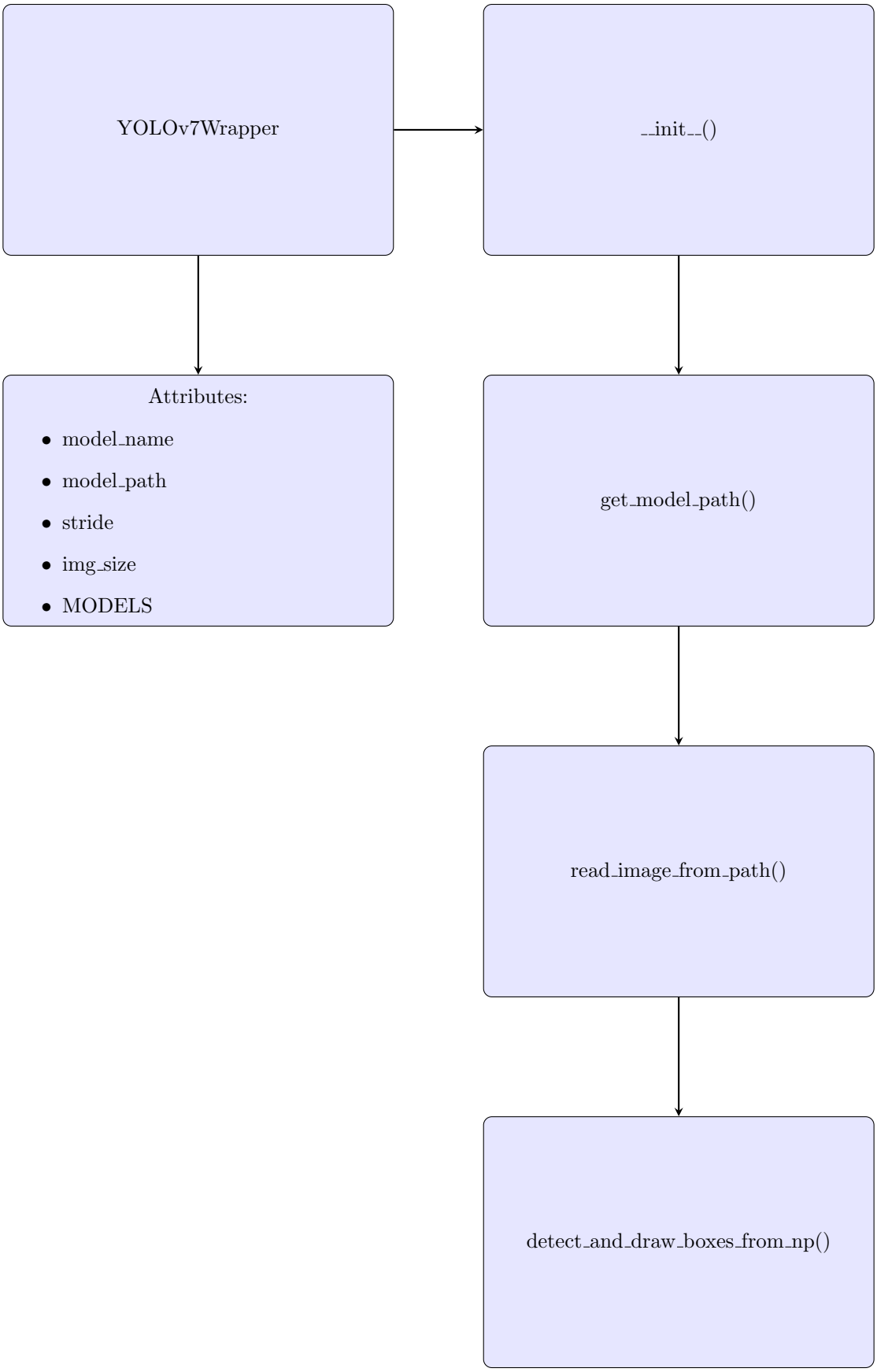


Figure 10: Structure and functionality of the YOLOv7Wrapper class.

4.3.2 Inconsistent Model Name Handling

Description During the implementation, there was an issue where the app only read the first letter of the model name, leading to errors in loading the correct model. This problem was significant in the `run_detection` and `detect_with_v7` functions.

Solution The solution involved ensuring consistent handling of the model name throughout the code. This was achieved by revising the functions to correctly interpret and pass the entire model name string rather than just the first character.

4.3.3 Memory Allocation Errors

The app encountered several `MemoryError`s like the following example during YOLOv7 inferences when loading and processing images, especially when switching between different models.

```
MemoryError: Unable to allocate 2.64 MiB for an array with shape (1280, 720, 3)
and data type uint8
```

This issue was related to inefficient memory usage in the image processing functions.

To resolve this, the application's image processing workflow was optimized. This included caching the processed images using Streamlit's caching capabilities ("Caching - Streamlit Docs", 2023) and ensuring proper memory management

Workflow Optimization To optimize the application's performance and resource management, the initial approach of using buttons for model execution was considered. Earlier, buttons in the Streamlit interface triggered the execution of each model separately. While this provided user control, it led to the entire Streamlit script rerunning, causing several app crashes. The workflow was adjusted to process images with different models sequentially to address this. This change smoothed the execution process, reducing the computational load by avoiding simultaneous model runs, thus enhancing overall stability and performance.

```python
# Before: Button-based model execution
if st.button(f'Run {model_name}'):
    # Run detection for Model 1
if st.button(f'Run {model_name}'):
    # Run detection for Model 2
```

(a) Initial button-based approach

```python
# After: Sequential model execution
for model_name in selected_models:
    # Run detection for each model sequentially
```

(b) Optimized sequential processing

Figure 11: Comparison of model execution approaches in Streamlit application

Streamlit Caching

Exploration of memory solutions led to considering caching strategies in Streamlit, per the platform's documentation ("Caching - Streamlit Docs", 2023).

A `CachedStFunctionWarning` was triggered in Streamlit due to the use of `st.code` within a cached function. This warning indicated potential issues with how the caching mechanism was being utilized.

The warning was addressed by moving the `st.code` call outside of the cached function. The caching decorator was also updated to use the more appropriate `st.cache_data` as per Streamlit's latest recommendations.

4.3.4 Handling Results from Different YOLO Models

During the development of our application, we faced significant challenges in processing and displaying the results from different versions of YOLO models (YOLOv7 and YOLOv8). These models, while similar in their objective of object detection, produced results in slightly varied formats, necessitating tailored approaches for handling and presenting these results.

Issues Encountered

- The primary issue was the inconsistency in the format of the detection results. While YOLOv8 results were straightforward to process, YOLOv7 results posed challenges due to differences in the structure of the output dictionary, particularly in the handling of the `count` key.
- The results from YOLOv7 models were nested in a complex dictionary structure, making it challenging to extract and display the detection counts and details effectively.
- Our application's goal was to allow users to compare the performance of different models side-by-side. This required aggregating results from multiple models, complicated by the mentioned inconsistencies.

Solutions Implemented To address these challenges, we implemented several strategies:

- We started by thoroughly analyzing the output structure of both YOLOv7 and YOLOv8 models. This involved printing and examining the output dictionaries to understand the keys and nested structures.
- We developed tailored functions (`update_v7_results` and `update_v8_results`) to handle the extraction of relevant data from the results of each YOLO version. These functions were designed to navigate through the nested dictionaries and extract the `count` and `details` data efficiently.

```
def update_v7_results(aggregated_results, results, model_name):
    if 'count' in results:
        for class_id, details in results["count"].items():
            if class_id not in aggregated_results:
                aggregated_results[class_id] = {}
            aggregated_results[class_id][model_name] = details['count']
```

```
def update_v8_results(aggregated_results, results, model_name):
    count = results.get("count")
    if count:
        for class_id, count_value in count.items():
            if class_id not in aggregated_results:
                aggregated_results[class_id] = {}
            aggregated_results[class_id][model_name] = count_value
```

- To aggregate the results for comparison, we implemented dynamic logic that could handle the different formats and update a unified results table. This approach ensured that even if a certain model did not detect any objects (resulting in an empty `count` key), the logic would still work seamlessly.

```
for selected_model_name, results in all_results.items():
    if selected_model_name in config.DETECTION_MODEL_LIST_V7:
        update_v7_results(aggregated_results, results, selected_model_name)
    else:
        update_v8_results(aggregated_results, results, selected_model_name)

# Displaying aggregated results
st.table(aggregated_results)
```

- To enhance the robustness of the application, we added comprehensive error handling to manage scenarios where the expected keys were missing in the results.

```
if 'count' not in results:
    st.error(f"Count data not found in results for model: {model_name}")
else:
    process_count_data(results['count'], model_name)
```

- Leveraging Streamlit's capabilities, we created interactive tables and detailed sections in the application's UI to display the aggregated results clearly and intuitively for user comparison.

As a result of these implementations, the application successfully handled and displayed the detection results from different YOLO models in a coherent and user-friendly manner. This allowed users to compare the performance of these models effectively, thereby achieving a key functionality of the application.

To summarise, though integrating YOLOv7 presented challenges, our methodical approach, supplemented by the versatility of the custom wrapper, enabled us to forge a robust and user-centric solution.

4.4 Visualization and Insights

The statistics section of the dashboard offers users insights into the inferences made using the YOLO models. It provides a comprehensive view of the data, visual representations, and other statistical information that are key to understanding the detection trends.

4.4.1 Statistics Dashboard Features

Data View

- The dashboard provides a detailed data table view, offering a complete look at the inference results.
- Users can download the complete dataframe as an Excel spreadsheet.
- Furthermore, a filter option allows users to select a specific value and see a filtered table based on that value. This filtered data can also be downloaded as an Excel file.

Summary Statistics

- Total number of inferences made.
- Number of unique objects detected.
- Average count of detections across all inferences.

Visualizations

Frequency of Detected Objects A bar chart showcases the frequency of each detected object. As shown in Figure 14 , this visualization offers a clear representation of which objects are detected more often.

Proportion of Detected Objects A pie chart illustrates the proportion of each detected object in relation to the whole dataset. Refer to Figure 12.

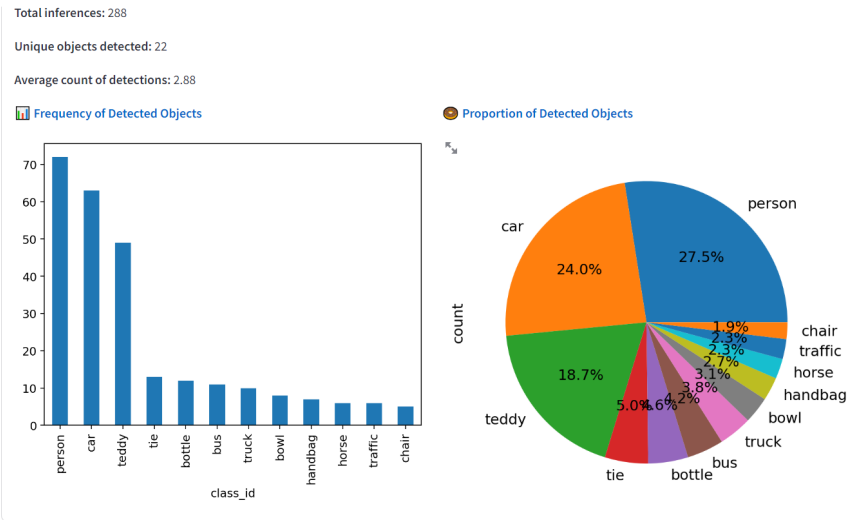


Figure 12: Summary Statistics

Analyzing Filtered Data In the following section, we delve into the graphical representations of the data post-filtration, providing key insights and interpretations that emerge from the visual analysis.

Historical Detection Analysis

- **Total Detections Over Time:** A line chart detailing how the number of detections has changed over time.

- **Unique Objects Detected Over Time:** A line chart showing the trend in the number of unique objects detected over time.
- **Temporal Trends in Detection Frequency:** This section includes a line chart illustrating the trend of average detection counts over various time periods. Refer to Figure 13 for a visual representation.

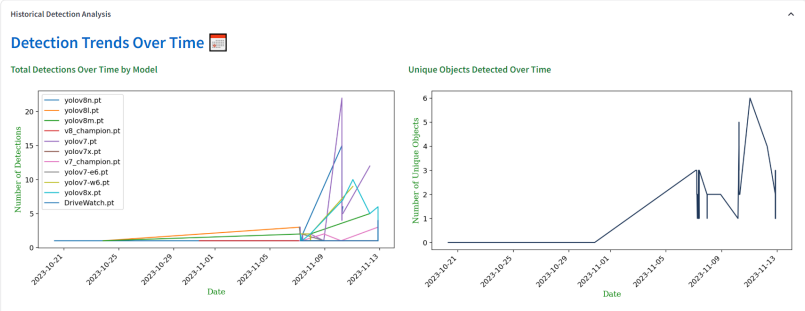


Figure 13: UI showing trends over time

4.4.2 Handling of Filtered Data and Visualization

The objective of handling filtered data was to provide tailored visualizations that reflect the specific parameters selected by the user. This included the ability to filter by model, class ID, and date range, with the goal of offering a more focused analysis of the inference data.

Data Filtering Approach The data was meticulously filtered based on user selections from a dynamic interface. The filtering logic was designed to be responsive to user input, ensuring that only the most relevant data was presented in the dashboard (As shown in Figure 14).

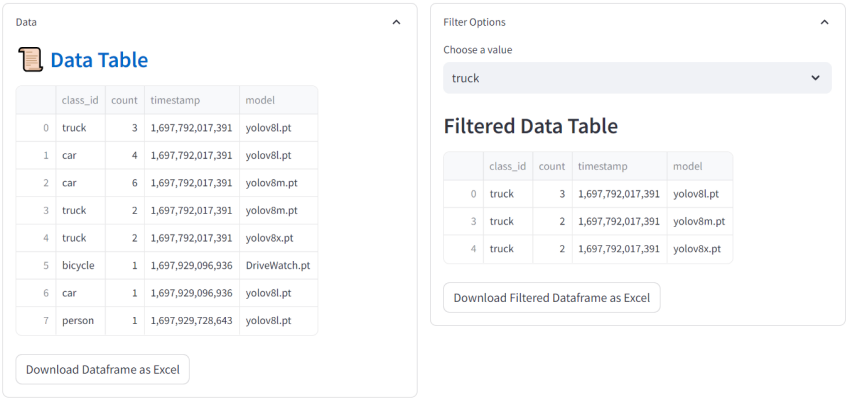


Figure 14

Visualization Strategy Visualizations were generated to translate the filtered data into easily digestible insights. Special attention was given to accurately represent the distribution and trends of the selected subsets of data. The visualization tools were chosen for their ability to highlight specific aspects of the data, aiding in the user's understanding of the underlying patterns and anomalies.

Results and Insights for Filtered Data

The application's results, processed from the refined data, provide insightful visualizations that emphasize the effectiveness of our data handling and visualization strategy. As depicted in Figure 15, the interface allows

users to meticulously filter the dataset based on specific parameters such as model selection and class IDs, enabling a granular analysis of object detection results.

The histogram on the left side of Figure 15 presents the confidence score distribution across various classes. It reveals the frequency of each confidence score, offering a clear view of the model's certainty in its predictions. For instance, a higher frequency of high confidence scores for a particular class, such as 'traffic', indicates a robust detection capability for that class by the model.

Conversely, the heatmap to the right aggregates the detection instances by model and class, providing a comparative perspective across the class IDs. The intensity of the color correlates with the number of detections, allowing for a quick assessment of which classes are most commonly detected by a given model. This not only assists in evaluating the model's performance but also highlights potential areas for further optimization.

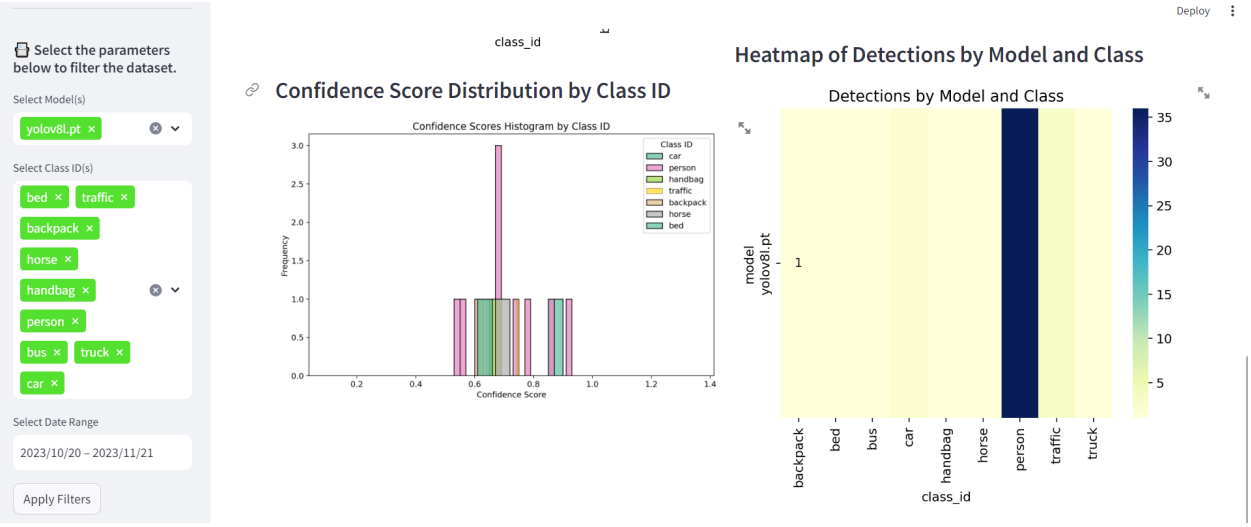


Figure 15: Visualizations of the model's detection performance, with the left histogram showing the distribution of confidence scores by class ID, and the right heatmap displaying the count of detections by model and class.

4.4.3 Challenges

During the process, several challenges were encountered, particularly in ensuring the clarity and accuracy of the visualizations. The primary issues revolved around the dynamic nature of the data filtering, which required robust error handling to manage instances of missing or incomplete data.

No Previous Data Found for User

But what if a user has no data stored in Realtime Database? One error message we saw was:

```
AttributeError: 'NoneType' object has no attribute 'values'
```

This error popped up when we tried to get some values from our data from Firebase.

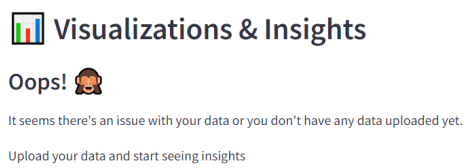
To fix this, we had to check the `get_inference_data(user_id)` function. We ensured that without data from Firebase, our function would know and handle it. This way,
our `visualize_inferences()` function could work smoothly, even if there was no user data. You can see an example of UI when there's no data in Figure 16.

The visualizations clearly show how well the model works and how good we are at presenting data. They make the model's results easy to understand and use, laying the groundwork for future enhancements and informed decisions.

In summary, the chosen method for visualizing data has been very effective. It has achieved and even exceeded our goals, making complex data easy to understand. We overcame the challenges in refining and presenting the data, resulting in a powerful tool that gives us a comprehensive and detailed view of the model's performance.

```python
def visualize_inferences():
    user_id = get_logged_in_user_id()
    if not user_id:
        st.warning("User not logged in or user ID not
        ↪  available.")
        return
    data = get_inference_data(user_id)
    if not data:  # Added data validation
        st.markdown("## Visualizations & Insights")
        st.markdown("### Oops!")
        st.write("It seems there's an issue with your data or
        ↪  you don't have any data uploaded yet.")
        st.write("Upload your data and start seeing insights")
        return
```

(a) Code



(b) Output

Figure 16: UI of Statistics page when the user has no data saved

4.5 Enhancing User Experience

4.5.1 Incorporating Lottie Animation

To enhance user engagement and provide a visually appealing experience on the application's landing page, Lottie animations were incorporated. Lottie animations are lightweight, scalable, and interactive, making them popular for modern web applications, including Streamlit (Streamlit Community, 2023).

Landing Page and Dashboard: As shown in Figure 18, the Lottie animation was placed on the landing page as an interactive placeholder. It remains visible until a user selects a file for processing. This approach enriches the user interface and provides a dynamic element to engage users while they navigate the initial options.

Model Comparison Page: Similarly, on the Model Comparison page, Lottie animation plays a crucial role. It is an intuitive indicator that the system is processing the user's request, running model comparisons, and awaiting the results. This visual cue helps manage user expectations about the processing time and enhances the overall interactive experience.

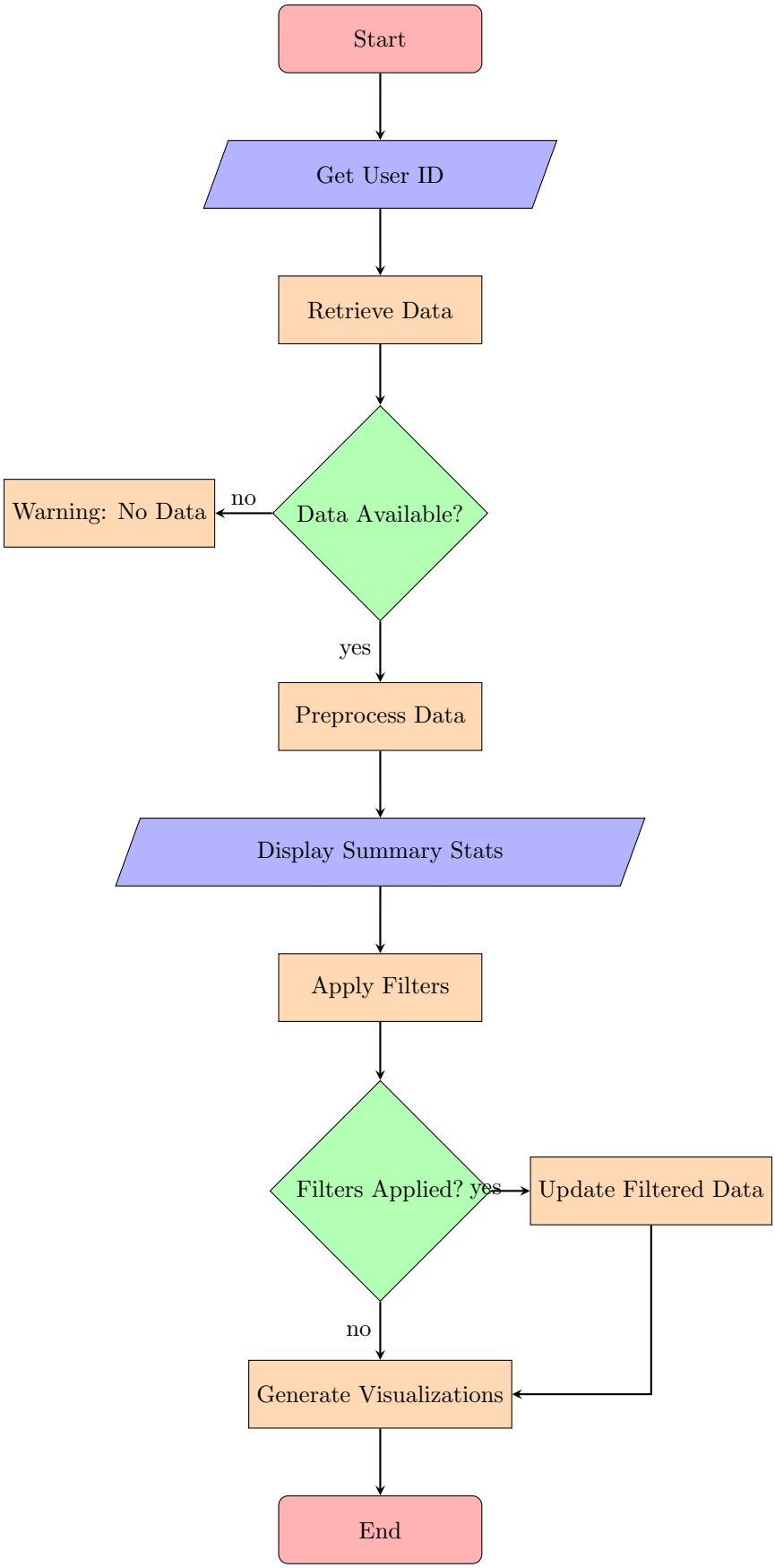


Figure 17: Process flowchart for the Statistics page operations.

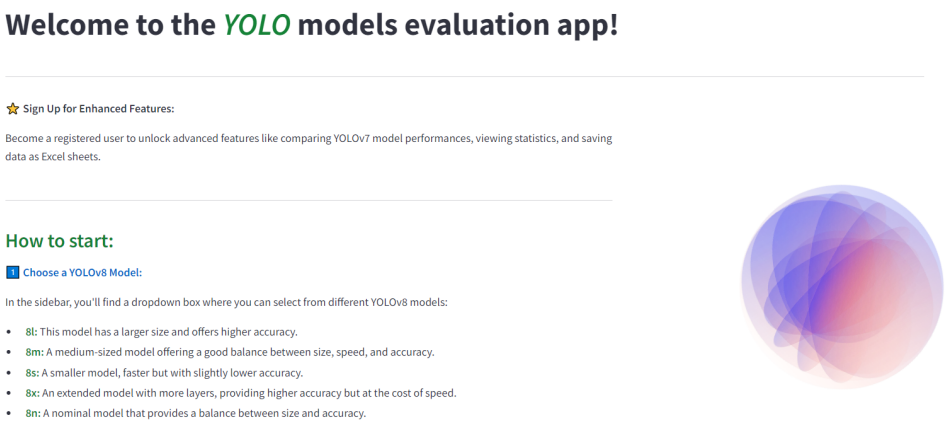


Figure 18: Screenshot of the Lottie Animation on the Landing Page

4.5.2 Incorporating Streamlit Emojis

To elevate the user experience and create a visually engaging interface within our Streamlit application, we integrated emojis in strategic locations across the user interface. Streamlit's native support for emojis allows for the injection of lively, relatable icons that resonate with users and add a layer of intuitive interaction (**streamlit˙emoji˙usage**).

Landing Page and Dashboard: As illustrated in Figure 19, emojis were employed on the landing page to visually represent the selection of machine learning models. For instance, the use of an emoji next to the "YOLOv8 Models" dropdown menu provides a quick visual cue, enhancing the user's ability to navigate the application effectively. The interactive nature of Streamlit emojis ensures that they do not impede the application's performance while simultaneously enriching the interface.

Model Comparison Page: On the Model Comparison page, emojis are used to offer a friendly and engaging means of presenting different models for comparison, as seen in Figure 20. These emojis serve as visual anchors for the users, making the selection process more intuitive and less monotonous. Their use in this context exemplifies how subtle graphical elements can significantly impact user experience, particularly in scenarios involving complex interactions like model selection.

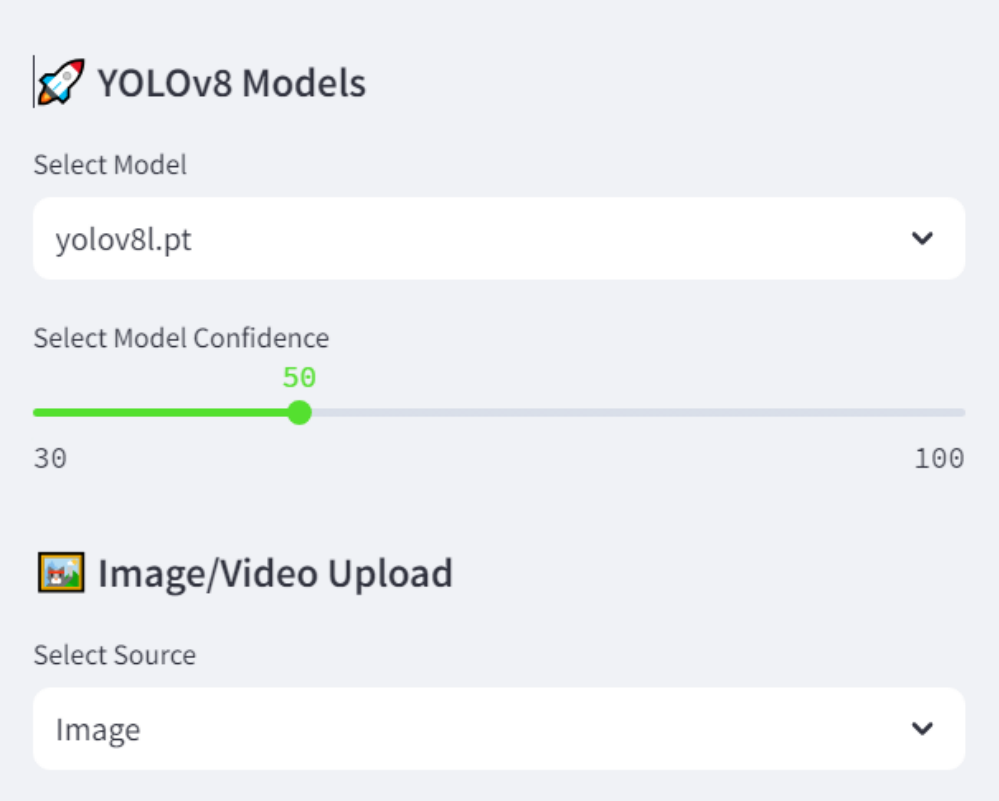


Figure 19: Use of Streamlit emojis on the Landing Page for model selection.

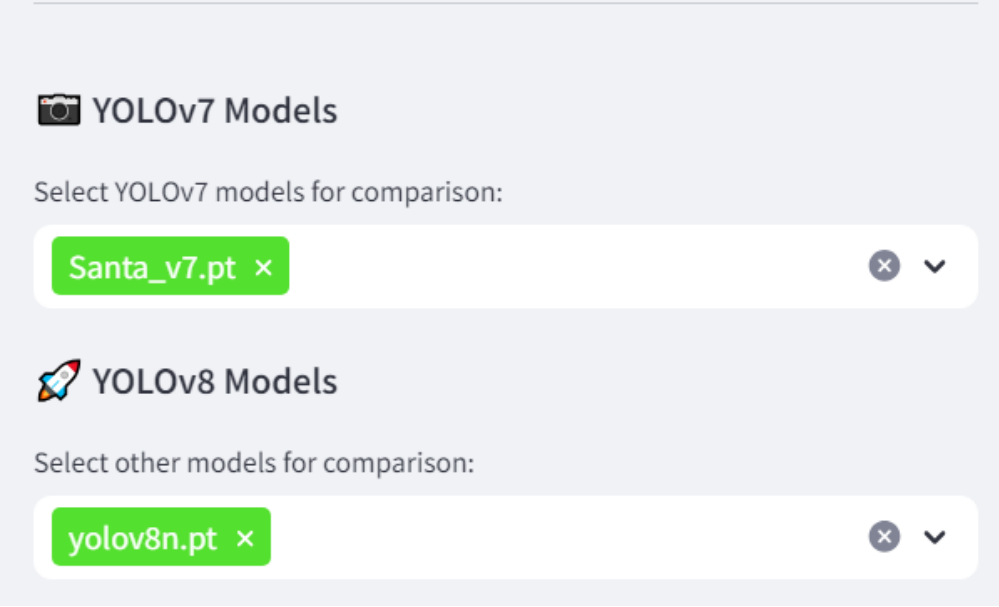


Figure 20: Streamlit emojis enhancing the Model Comparison selection interface.

Key Takeaways The integration of YOLOv7 and YOLOv8 models into our application marks a significant milestone in our journey towards advanced object detection. This undertaking has not only enhanced the accuracy and efficiency of our tool but also provided valuable insights into the complexities of modern

machine learning models.

Throughout this process, we faced numerous challenges, from model compatibility issues to data handling complexities. Each challenge was met with innovative solutions, resulting in a robust and user-friendly application. The experience gained from this project has been immense, providing a strong foundation for future enhancements and expansions in the realm of computer vision and object detection.

5 More Challenges and Solutions

5.1 Improving Firebase Security

Firebase, a Backend-as-a-Service (BaaS) provided by Google, offers a real-time database among its many services. As with any cloud service, it is ensuring data security and privacy is paramount. This report details the modifications to the Firebase Realtime Database security rules and the reasons behind these changes.

Originally, the security rules permitted user data access based solely on the user's unique identifier (UID). The user could read or write to their specific data node, and there were basic validations for the name and email properties.

The initial rule set, though user-restricted, lacked a few essential features:

- A default restrictive policy for all database nodes.
- Comprehensive validation for email data.
- Protection against potential additional data nodes.

To enhance security, a principle of least privilege was adopted. The rules were updated to deny read and write access at the root level by default. Any new data node or path introduced will be inaccessible unless explicitly defined in the rules.

```
 [language=json, caption=Top-level Default Denial]
{
  ".read": false,
  ".write": false
}
```

While the original rules had some validation in place for name and email fields, the email validation was basic, merely checking for the presence of an '@' character. Though comprehensive email validation on the database side can be challenging due to various valid email formats, the validation was slightly improved for better security.

```
[language=json, caption=Email Validation]
"email": {
  ".validate": "newData.isString() && newData.val().contains('@')"
```

5.2 Git

This section documents the challenges and solutions related to Git version control encountered during development. The primary issues revolved around branch management and proper synchronization between local and remote repositories.

5.2.1 Branch Management and Synchronization Issues

Several issues were encountered with managing branches and synchronizing changes between the local and remote repositories during the development process. These issues were critical in maintaining the consistency and integrity of the project's codebase.

Problems Encountered The following Git-related issues were encountered:

- Inability to push to the `main` branch because it did not exist locally.
- Non-fast-forward errors during push attempts, indicating the need for local synchronization with the remote `main` branch.
- Merge conflicts when synchronizing the local and remote `main` branches.

Solutions Implemented To resolve these issues, the following steps were undertaken:

1. Creation and checkout to a new local `main` branch to align with the remote repository structure.
2. Execution of `git pull` to fetch and merge changes from the remote `main` branch.
3. Manual resolution of merge conflicts in conflicting files.
4. Staging and committing the resolved changes.
5. Successful push of the updated local `main` branch to the remote repository.

Command Log The command log below details the specific Git commands used in the process of troubleshooting and resolving the issues:

```
# Check current branch
git branch

# Rename local branch to main
git branch -m master main

# Pull changes from remote main
git pull origin main

# Push changes to the remote main
git push origin main

# Resolve merge conflicts
git add <file>
git commit -m "Resolved merge conflicts"
git push origin main
```

The resolution of these issues ensured that the local and remote repositories were properly synchronized, with the `main` branch reflecting the project's most current and stable version.

5.2.2 Correcting VS Code Account Setup

Initial confusion occurred because Visual Studio Code was initially set up with a secondary account. To ensure commits were attributed to the primary account, the Git configuration was updated as follows:

```
# Set the email address for Git
git config --global user.email "irinagetman1973@gmail.com"
```

These commands updated the global Git configuration, ensuring that all future commits in VS Code were made with the correct user credentials.

5.2.3 Managing Multiple Branches

Multiple branches due to pull requests, pushing to `master` instead of `main`, all added up to a rather messy repository. The following steps were taken to manage and synchronize these issues:

```
# Deleting a remote branch
git push origin --delete irinagetman1973-patch-7
```

Care was taken to ensure no data was lost during this cleanup process.

5.3 Deployment on AWS

Deploying our sophisticated application on Amazon Web Services (AWS) involved a sequence of deliberate and strategic actions to ensure both the application's robust functionality and its security.

1. **Setting Up an EC2 Instance:** The deployment journey commenced with the selection of an Amazon Machine Image (AMI), specifically the AWS Linux/UNIX platform. We opted for the g4dn.2xlarge instance type, equipped with 8 CPUs — a choice informed by the intensive computational needs of Machine Learning applications. A configuration of security group rules was also established to protect the instance, complemented by the creation of a key pair to facilitate secure access.

2. **Securing the Private Key File:** In the Windows operating system, it is paramount to safeguard the private key file. This was achieved by modifying the file's permissions to read-only, as depicted by the following commands:

```
icacls C:\path\to\302.pem /inheritance:r
icacls C:\path\to\302.pem /grant:r "%username%:R"
```

3. **Establishing a Connection to the Instance:** Our digital gateway to the instance was established through the secure SSH protocol, with the command structure as follows:

```
ssh -i "C:\path\to\302.pem" ec2-user@instance-public-ip
```

4. **Application File Transfer:** Employing SCP, we securely migrated the application files to the instance, ensuring integrity and confidentiality. The command executed was as follows:

```
scp -i "C:\path\to\302.pem" C:\path\to\capstone.zip
ec2-user@instance-public-ip:~/
```

5. **Application Deployment:** Post file transfer, we embarked on setting up the application's environment on the EC2 instance. This encompassed installing dependencies and initiating the application:

```
streamlit run main.py
```

Subsequent to these meticulous steps, our application was successfully deployed and operational on the EC2 instance, actively listening on the default HTTP and HTTPS ports as defined by the instance's security groups.

During the initial testing phase within the instance, we encountered a series of challenges:

1. The Streamlit application necessitated an update.

2. Our Python environment lacked the libGL library, an essential component for OpenCV's functionality, indicated by the `import cv2` statement. This issue is a common pitfall when executing graphical libraries on headless servers, such as AWS EC2 instances.

   Solution:

```
# Install libGL on Amazon Linux
sudo yum install -y libglvnd-glx
```

3. Encountering a `CalledProcessError`, we diagnosed that the instance required Git tags, leading to a pivot in our deployment strategy.

Addressing the need for a Git tag, we executed the following:

```
# Install Git on the Amazon Linux instance
sudo yum install git -y

# Initialize and update submodules
git submodule init
git submodule update

# Create a Git tag
git tag v1.0.0
git push origin v1.0.0
```

The evolution from an initial file upload to the cloning of a Git repository brought a robust solution of dependency management, particularly for submodules like `yolov7`. This strategy proved instrumental in ensuring a successful development environment and streamlined the code and dependency update processes. The following commands illustrate the cloning of the repository and the initiation of submodules:

```
# Clone the repository
git clone https://github.com/irinagetman1973/YOLO-Streamlit

# Navigate to the repository directory
cd YOLO-Streamlit

# Initialize and update submodules
git submodule update --init --recursive
```

Given that our YOLO models were excluded from version control via the .gitignore file, we resorted to the SCP method once more to transfer these essential elements of our application from the local machine to the EC2 instance. With the repository and submodules now in place, subsequent updates could be effortlessly pulled from the Git repository, ensuring up-to-date submodule references.

```
# Fetch the latest changes
git pull

# Update the submodules
git submodule update --init --recursive
```

This refined process shines in dynamic environments where frequent codebase updates are the norm, and stringent dependency management is of the essence.

As evidenced in Figure 21, the application was accessible via the public IP address, ensuring a smooth user experience.

In this section, we navigated through a myriad of technical challenges and their solutions, crucial for the project's success. Enhancements in Firebase security rules were implemented to ensure robust data protection. Git version control posed its own set of challenges, skillfully addressed through systematic branch management and synchronization strategies. The deployment phase on AWS highlighted the importance of environment-specific configurations and problem-solving. Each hurdle encountered was an opportunity to refine our approach, leading to a more resilient and scalable application. These experiences not only resolved immediate issues but also laid a foundation for future advancements in the project.

6 Results and Analysis

In our comparative analysis of YOLOv7 and YOLOv8, we observed distinct performance characteristics that align with the specific strengths of each model. YOLOv7 demonstrated high accuracy in object detection, albeit with a slightly lower processing speed. This precision makes it ideal for scenarios where accuracy is paramount. However, integrating YOLOv7 presented more complexities, necessitating additional workarounds to harmonize its functionality with the rest of the application.

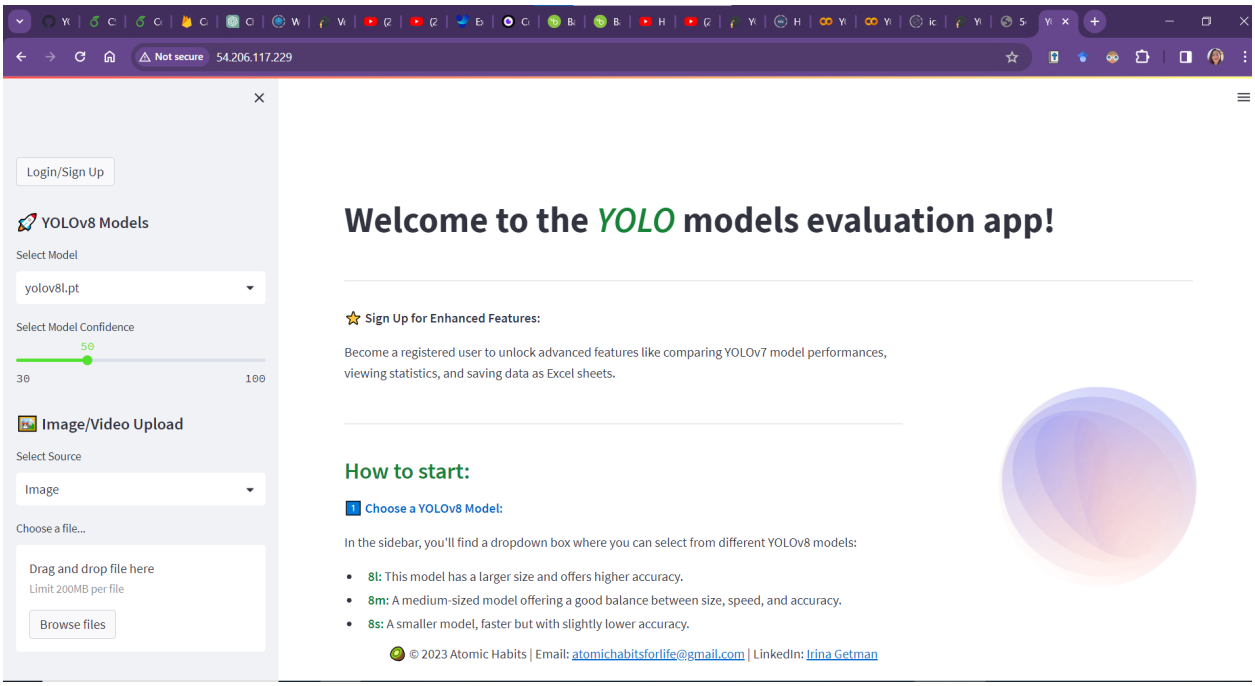


Figure 21: Screenshot of the deployed application accessible via the IP address.

On the other hand, YOLOv8 excelled in speed, offering real-time detection capabilities, thus proving to be more suitable for applications requiring rapid response, such as real-time surveillance. YOLOv8's user-friendly API further contributed to a smoother integration process, enhancing the overall user experience.

The user interface, designed to be intuitive and interactive, facilitated seamless user engagement with both models. Feedback from test users highlighted the ease of model selection and the clarity of the visual comparison feature. This indicates that our design choices effectively bridged the gap between complex machine-learning operations and user-friendly interaction.

7 Discussion

The project's trajectory, although initially ambitious, was recalibrated to focus on the integration of YOLOv7 and YOLOv8 models. This decision, dictated by practical constraints and technical challenges, ultimately led to a more refined and effective application. The reduced scope allowed for a deeper exploration into the nuances of these two models, providing users with a valuable tool for object detection tasks.

Reflecting on the project's objectives, we successfully created an application that not only compares the performance of YOLOv7 and YOLOv8 but also provides an accessible platform for users to explore advanced object detection technologies and access valuable insights. This achievement underscores our ability to adapt and evolve in response to the dynamic nature of technology development.

8 Conclusion and Future Work

In conclusion, our project stands as a testament to the practical application of cutting-edge machine learning models in a user-friendly format. The successful integration of YOLOv7 and YOLOv8 into our application provides users with powerful tools for object detection, coupled with an intuitive interface for ease of use.

Looking ahead, the potential for future expansions includes incorporating additional models, enhancing the application's analytical capabilities, and exploring the integration of real-time data streaming. Our journey in the realm of computer vision and object detection is far from over, and we eagerly anticipate the advancements and innovations that lie ahead.

References

Caching - streamlit docs [Accessed: 1/11/2023]. (2023).
Smiley, S. (2022). *Train deploy yolov7 to streamlit* [Accessed: 10/10/2023]. https://stevensmiley1989.medium.com/train-deploy-yolov7-to-streamlit-5a3e925690a9
Streamlit Community. (2023). Lottie animation in streamlit [Accessed: 9/11/2023].