BAHRIA UNIVERSITY
PAKISTAN

DEPARTMENT OF
COMPUTER SCIENCES
ISLAMABAD CAMPUS

DEPARTMENT OF COMPUTER SCIENCES

Software Engineering

Title: **Human Action Recognition System**

SUBMITTED BY:

Muhammad Awab Sial (01-134222-091)

Syed Amber Ali Shah (01-134222-146)

Fawad Naveed (01-134222-049)

Contents

Chapter 1: Introduction

1.1. Introduction

Artificial Intelligence (AI) has revolutionized the way we interact with technology, offering unprecedented capabilities in automation, personalization, and user engagement. In recent years, advancements in computer vision and artificial intelligence have enabled machines to understand and interpret human actions from visual data. This capability has applications in diverse fields, including security and surveillance, sports analytics, healthcare monitoring, human-computer interaction, and more. For instance, security systems can detect suspicious activities in real time, while healthcare systems can monitor patient movements to prevent falls or ensure rehabilitation compliance.

Similarly, human action detection continues to evolve, with trends focusing on transformer-based architectures, multimodal approaches, and lightweight models for real-world applications.

Despite these advancements, several challenges persist:

- Complexity in Recognition: Existing systems struggle with recognizing actions in varied environments with changing lighting, backgrounds, and occlusions.
- Real-Time Processing: Many systems fail to provide real-time feedback due to computational bottlenecks.
- Accessibility: Current solutions often lack user-friendly interfaces, making them inaccessible to non-technical users or small-scale organizations.

With ongoing trends in transformer-based architectures, multimodal approaches, and lightweight models, human action detection continues to evolve, paving the way for realworld applications that are efficient and inclusive.

1.2. Existing Solutions & Their Limitations

1. AlphaPose

- **What it Does:** A high-performance system for pose estimation and human action recognition. It identifies body key points and interprets actions using the pose data.

- **Applications:** Sports analysis, behavior monitoring, and gesture recognition.

2. DeepMind Kinetics

- **What it Does:** A large-scale action recognition dataset used to train systems for recognizing human actions in videos. Models trained on Kinetics can identify hundreds of human actions.

- **Applications:** General-purpose action recognition tasks in videos.

3. MoViNet (Efficient Video Networks)

- **What it Does:** Designed for efficient human action recognition, it processes video frames to detect actions in real-time on mobile devices.

- **Applications:** Real-time applications like fitness tracking and mobile action detection.

4. PoseC3D

- **What it Does:** Focuses on recognizing human actions using skeleton data from pose estimation algorithms instead of RGB data.

- **Applications:** Privacy-focused action recognition in smart homes or workplaces.

5. MMAction2 (OpenMMLab)

- **What it Does:** A modular and scalable framework for video action recognition tasks. It includes various pre-trained models and methods like TSN, TSM, and SlowFast.

- **Applications:** Video surveillance, content tagging, and autonomous systems.

6. Two-Stream Inflated 3D ConvNet (I3D)

- **What it Does:** Combines spatial (image frames) and temporal (motion flow) features using 3D convolutional networks for action recognition in videos.
- **Applications:** Sports analytics, security surveillance, and video summarization.

7. HARNet (Human Activity Recognition Network)

- **What it Does:** A neural network designed for recognizing human activities using wearable sensor data or video sequences.
- **Applications:** Health monitoring and fitness tracking.

8. OpenPose

- **What it Does:** While primarily a pose estimation system, OpenPose can be combined with classification models to recognize human actions based on skeletal data.
- **Applications:** Gesture-based gaming, rehabilitation, and social robot interaction.

9. ST-GCN (Spatial-Temporal Graph Convolutional Networks)

- **What it Does:** Uses graph-based modeling of human skeletons over time for action recognition from pose sequences.
- **Applications:** Health and fitness monitoring, sports training, and interactive systems.

10. Google Action Transformer (GAT)

- **What it Does:** Leverages transformer models for temporal understanding of actions in videos, excelling at capturing complex sequences of actions.
- **Applications:** Video analysis, event detection, and video search engines.

Limitations:

Real-Time Constraints: Many of these models struggle with real-time processing, especially on low-resource devices.

Generalization: Accuracy often decreases when applied to diverse environments or unseen datasets.

Complexity: Some systems are challenging to set up or use without technical expertise.

Data Dependency: Models are heavily dependent on large and diverse datasets for training, which may not always be available.

Occlusions and Noise: Many systems fail to perform well in cases of occlusion, poor lighting, or noisy backgrounds.

1.3. Scope

The primary scope of this project is to develop a scalable and efficient Human Action Recognition System that bridges the gap between technical complexity and user accessibility. The following measurable goals guide the project:

1. **AI Model Development:**

   o Train a deep learning model for classifying human actions in standard datasets for 15 different classes

   o Optimize the model to ensure real-time performance for live video streams.

2. **User-Friendly Front-End:**

   o Develop an intuitive interface allowing users to upload videos/images, view results, and interact with the system effortlessly. o Include visual tools for presenting classification results and performance metrics.

3. **Integration with Hardware (Pc and Smartphone Camera)**

   o Achieve compatibility with PC cameras and through smartphones with real-time human activity recognition accuracy maintained.

Out of Scope:

1. Custom Action Sets: While retraining is supported, custom action sets beyond the predefined ones will require external development effort.

2. Advanced Analytics: Complex behavioral analysis or predictions based on multiple actions over time are beyond this project's scope.

3. Multi-Sensor Integration: The system will not integrate additional sensors like LiDAR or motion sensors.

1.4. Objectives

The objective of this AI-powered Human Action Recognition project encompasses the development of a robust, scalable, and user-friendly system capable of identifying and classifying multiple human actions from visual data. The system will leverage deep learning techniques to process video feeds or images, supporting real-time recognition with high accuracy. It will accommodate a range of predefined actions such as running, sitting, dancing, and more, while also enabling live action recognition from camera feeds. The project aims to provide an intuitive front-end interface for uploading videos or images, visualizing recognition results, and exporting them in formats like CSV or JSON for further analysis.

1.5. Dataset Specifications

Human Action Recognition (HAR) is a vital field in computer vision, enabling machines to identify and interpret human activities. This dataset, designed for HAR tasks, includes 12,600 labeled images of 15 common actions: calling, clapping, cycling, dancing, drinking, eating, fighting, hugging, laughing, listening to music, running, sitting, sleeping, texting, and using a laptop. It is split into training and testing sets with an 85-15% ratio to facilitate model development and evaluation.

The dataset aims to train systems that predict human actions from images, focusing on routine activities while acknowledging the impossibility of covering all human actions. It is suitable for developing deep learning models like CNNs for applications in surveillance, sports analytics, and human-computer interaction. With its structured format and clear labeling, the dataset provides a strong foundation for advancing HAR technologies and building systems capable of real-time action recognition.

- **Dataset name:** Human Action Recognition Dataset
- **Source:** The dataset is hosted on Kaggle by the user **Shashank Rapolu**
- **Format:** The dataset contains image sequences organized into folders corresponding to different action classes. All images are in .jpg format
- **Resolution:** random image resolutions from 128p to 720p
- **Classes:** The dataset contains image label generation on 15 activity classes:

1. Running
2. Sitting
3. Fighting
4. Dancing
5. Calling
6. Clapping
7. Cycling
8. Drinking
9. Eating
10. Hugging
11. Laughing
12. Listening to music
13. Sleeping

14. Texting

15. Using a laptop

⇨ **Images:** Total 12600 images

- 10710 training images
- 1890 testing images

⇨ **Data Augmentation used:**

- rescale=1.0/255.0,
- shear_range=0.2,
- zoom_range=0.2,
- rotation_range=30,
- width_shift_range=0.3,
- height_shift_range=0.3,
- brightness_range=[0.8, 1.2],
- horizontal_flip=True

1.5.1. Features of Dataset

1. **Action Classes:** Different types of human actions (e.g., walking, running, jumping, etc.).

2. **Labels:** Each video or image sequence is labeled with the corresponding action.

3. **Diversity:** Includes variations in:
   - Backgrounds.
   - Lighting conditions.
   - Performer demographics.
   - Action execution styles.

4. **File Metadata:** Videos may include information like frame rate, duration, and resolution.

5. **Split Availability:** Potential train/test/validation splits provided or easily configurable.

1.5.2. Strengths of HAR Dataset

- **Action Diversity:**

  The dataset likely contains a variety of human actions, making it

  suitable for training models to recognize multiple distinct activities, which is essential for generalized human action recognition systems.

- **Ease of Access and Organization:**

  The dataset appears well-structured with clear labels and organized folders for each action class. This makes it easier for researchers and developers to preprocess and use the data without significant overhead.

- **Versatility in Application:**

  Due to its labeled nature, it can be used for various machine learning models and applications, such as sports analytics, surveillance, and human-computer interaction systems.

- **Resource-Efficient:**

  The dataset's potentially low resolution and smaller file sizes could reduce computational requirements, making it suitable for researchers with limited hardware resources.

1.5.3. Weaknesses of HAR Dataset

- **Potential Lack of Real-world Variability:**

  The dataset might have limited environmental diversity (e.g., consistent lighting or background settings), which could make the trained models less robust in real-world scenarios where conditions vary greatly.

- **Resolution Constraints:**

  The relatively low resolution of the data (as observed in the sample image, 300x168) may restrict its effectiveness for high-precision tasks where finer details of human posture or movement are important.

- **Limited Performer Diversity:**
  If the dataset contains only a small set of actors or demographic groups performing the actions, it may lead to biases in the model and reduced generalizability across broader populations.

- **Lack of Annotations for Fine-grained Analysis:**
  Beyond labels, the dataset might not provide additional annotations (e.g., bounding boxes, skeleton/keypoint data) that could enable fine-grained analysis or enhance model interpretability.

1.5.4. Dataset Samples

1. **Calling:**

2. **Clapping:**

3. **Cycling:**

4. **Dancing:**

5. **Drinking:**

6. **Eating:**

7. **Fighting:**

8. **Hugging:**

9. **Laughing:**

10. **Listning to Music:**

11. **Running:**

12. **Sitting:**

13. **Sleeping:**

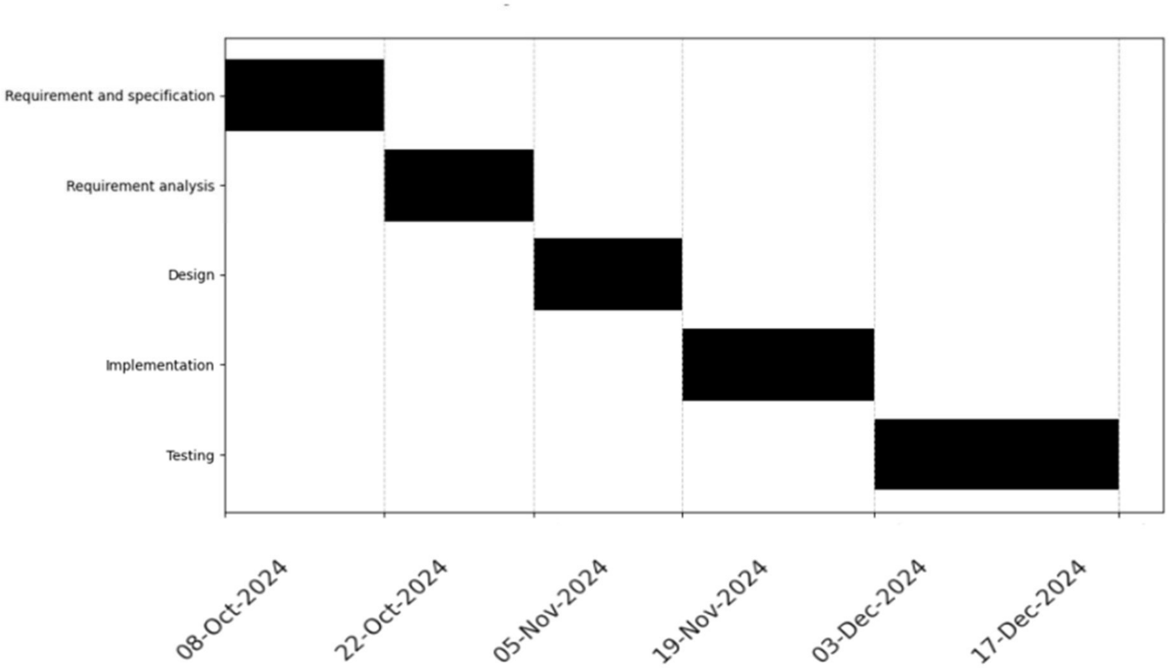
14. **Texting:**

15. **Using Laptop:**

1.6. Useful Tools & Technologies

Technologies	Purpose	Working
Python	Backend programming language	Used for integration of AI model and other components in the system.
HTML, CSS	Front end development	Creates an intuitive user interface for interaction and display of outputs.
Opencv2	Image and video processing	Captures and processes frames from video feeds, extracting features needed for action detection.
Keras -> sequential , layers	Deep learning framework	Builds the neural network architecture for training the human action recognition model.
Tensorflow	Machine learning library	Provides the backend for model training, inference, and optimization.
Numpy	Numerical computation	Handles matrix operations and data preprocessing for model input.
Hardware (Camera)	Input device for live video feed	Captures real-time video feeds or images to be processed by the AI system.
M1 processor	Hardware platform with building apple gpu	Provides computational power for efficiently running deep learning models and video processing.

1.7. Project Timeline



Chapter 2: Requirement Specifications

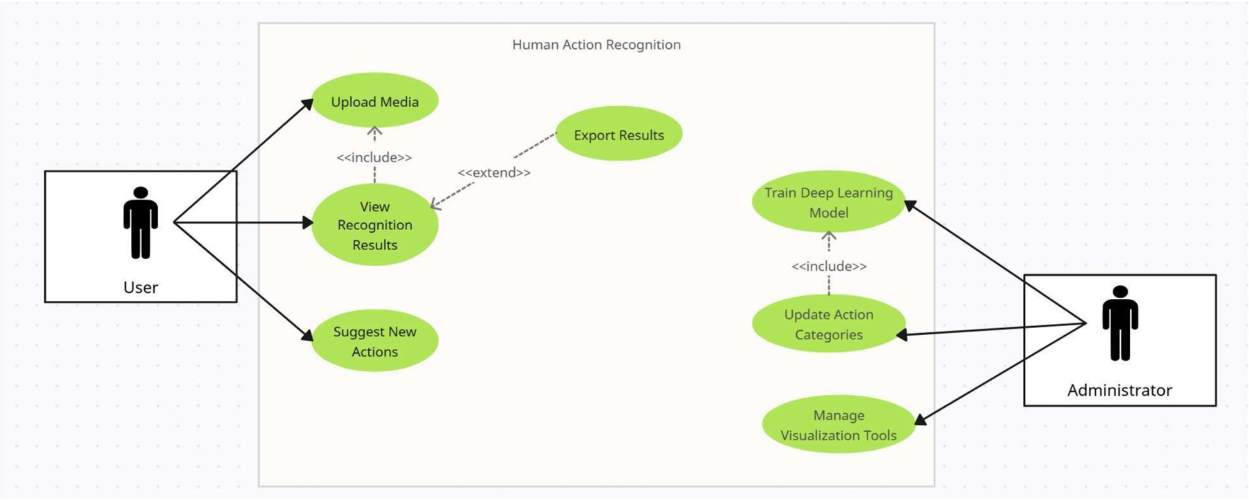
2.1. Functional Requirements

S. No.	Functional Requirement	Type	Status
1	Train a deep learning model to classify human actions from image data.	Core	Complete
2	Support multiple predefined action classes such as running, sitting, fighting, dancing, calling, clapping, cycling, drinking, eating, hugging, and others.	Core	Complete
3	Preprocess video frames to resize, normalize, and standardize inputs.	Core	Complete
4	Enable real-time action recognition from live camera feed.	Core	Complete
5	Provide an interface to upload videos/images for action recognition.	Core	Complete
6	Display real-time recognition results with visual feedback such as labels and percentage accuracy.	Core	Complete
7	Allow exporting of recognition results in user-friendly formats such as CSV or JSON.	Intermediate	Incomplete
8	Provide visualization tools like graphs for performance tracking.	Intermediate	Incomplete
9	Provide a mechanism to suggest new action categories.	Intermediate	Incomplete

2.2. Non-Functional Requirements

S. No.	Non Functional Requirement	Category
1	The system must provide a fast and efficient human action recognition response, processing images or video in real-time with minimal delay.	Performance
2	The system should consistently perform with accuracy and stability, minimizing loss and handling errors gracefully and ensuring minimal downtime during operation.	Reliability
3	The model should achieve an accuracy rate of at least 50% for human action recognition from both static images and video, with a low false-positive and falsenegative rate.	Accuracy
4	The system should be easily maintainable with modular code, enabling updates, bug fixes, and feature expansions without significant downtime.	Maintainability
5	The software interface should be intuitive and userfriendly for both technical and non-technical users.	Usability

2.3. Use-Case Modeling



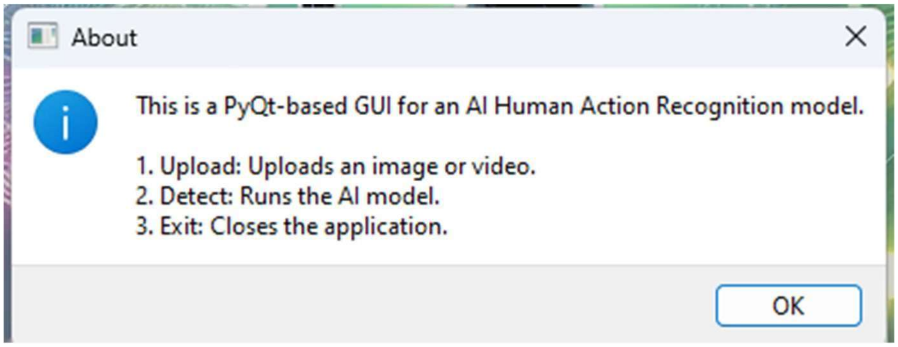
2.3.1. Use Case Ids

Use Case ID: 1001

Use Case Name:	Upload Button		
Created By:	Awab Sial	Last Updated By:	Awab Sial
Date Created:	12/22/2024	Last Revision Date:	12/22/2024
Actors:	User		
Description:	Allows the user to upload the media for detection		
Trigger:	User starts the detection		
Preconditions:	Application must be launched		
Postconditions:	The media to be detected will be uploaded		
Normal Flow:	Actor	System	
	Enters the media	Checks the validity of media type	
Alternate Flows:	If selected media type is not from allowed types, an error message will be displayed, and user will be re-prompted to select the media		

Use Case ID: 1002

Use Case Name:	Detect Button		
Created By:	Awab Sial	Last Updated By:	Awab Sial
Date Created:	12/22/2024	Last Revision Date:	12/22/2024
Actors:	User		
Description:	Allows the user to detect the action of the previously uploaded media		
Trigger:	User starts the detection		
Preconditions:	Media must be uploaded beforehand		
Postconditions:	The detection will be performed		
Normal Flow:	Actor	System	
	Wait for results	Detect the action and displays it	
Alternate Flows:	N/A		



This is a PyQt-based GUI for an AI Human Action Recognition model.

1. Upload: Uploads an image or video.
2. Detect: Runs the AI model.
3. Exit: Closes the application.

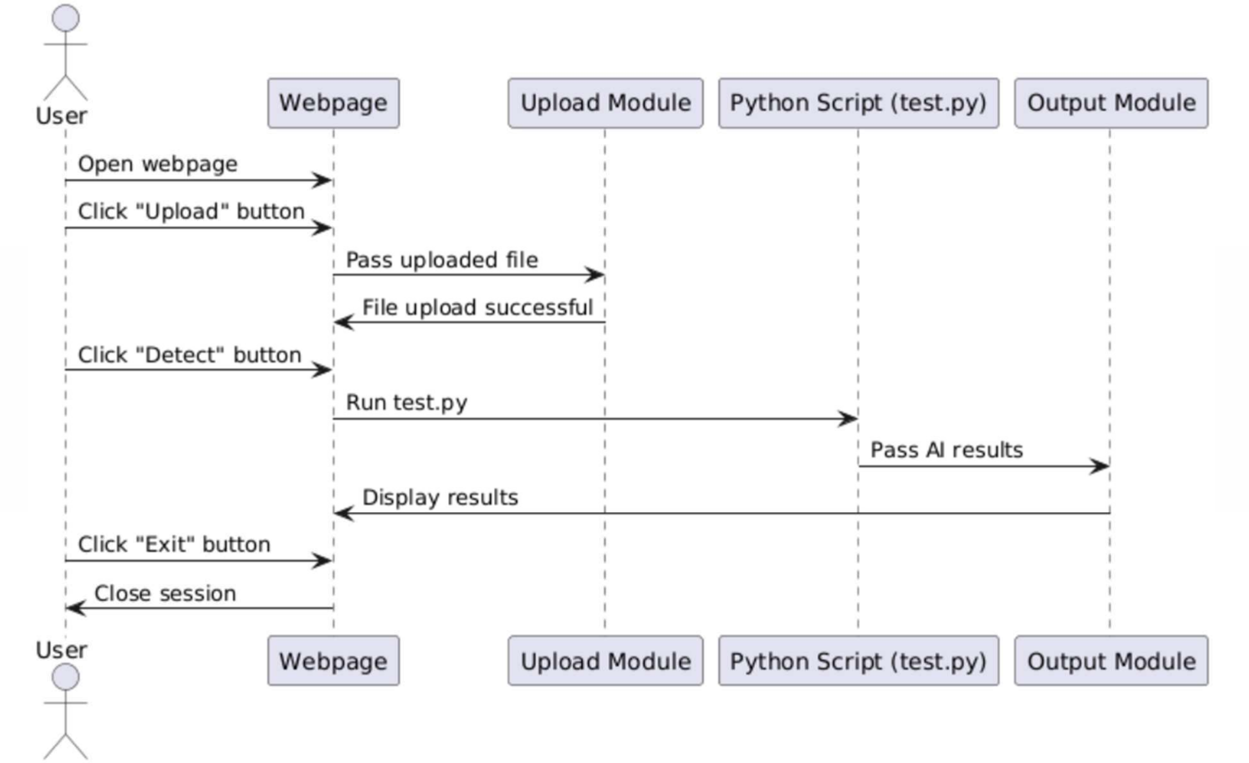
Use Case ID: 1003

Use Case Name:	Help Button		
Created By:	Awab Sial	Last Updated By:	Awab Sial
Date Created:	12/22/2024	Last Revision Date:	12/22/2024
Actors:	User		
Description:	Allows the user to understand the workflow of the application		
Trigger:	Launches a guide to navigate throughout the system		
Preconditions:	Application must be launched		
Postconditions:	Help message will be shown		
Normal Flow:	Actor	System	
	Message displayed	Help Message displayed	
Alternate Flows:	N/A		

Use Case ID:	1004		
Use Case Name:	Exit Button		
Created By:	Awab Sial	Last Updated By:	Awab Sial
Date Created:	12/22/2024	Last Revision Date:	12/22/2024
Actors:	User		
Description:	Allows the user to quit the application		
Trigger:	Quits the application		
Preconditions:	Application must be launched		
Postconditions:	Application will be closed		
Normal Flow:	Actor	System	
	N/A	Stops the current work and exits	
Alternate Flows:	N/A		

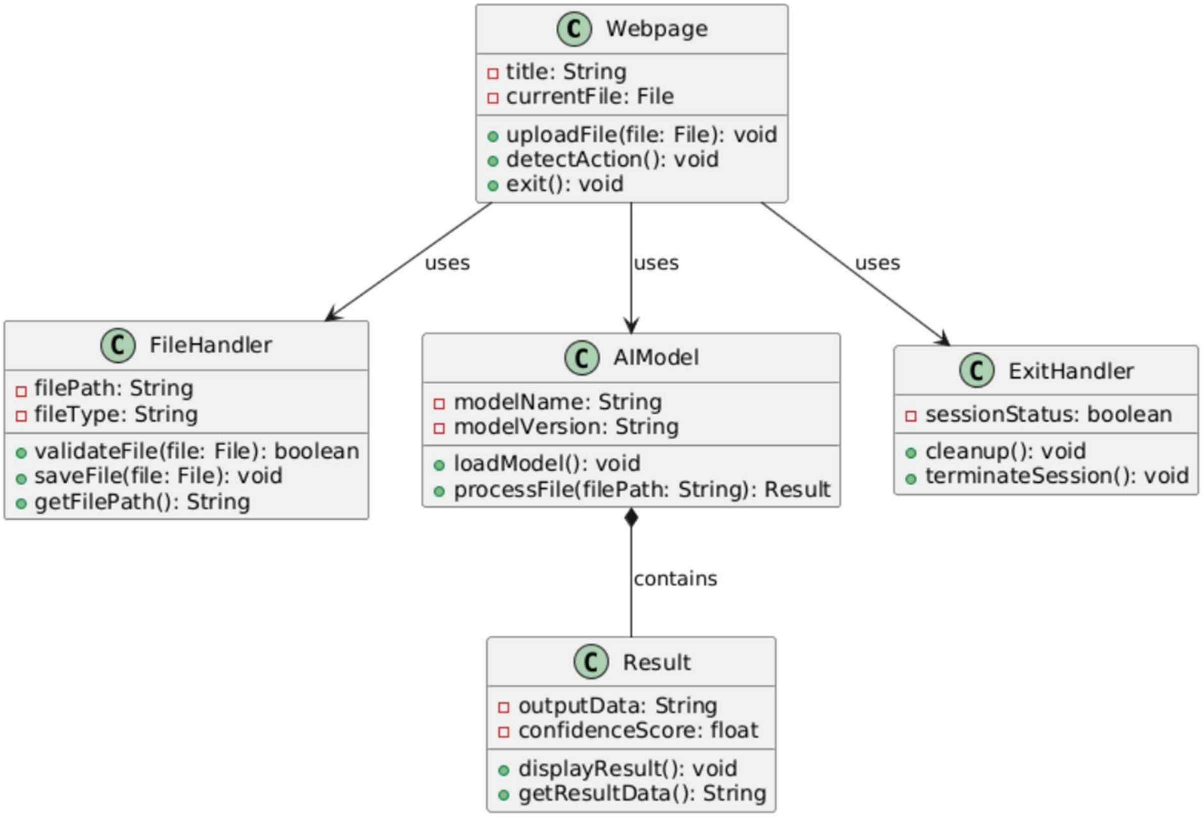
Chapter 3: Software Design Documentation

3.1. Sequence Diagram

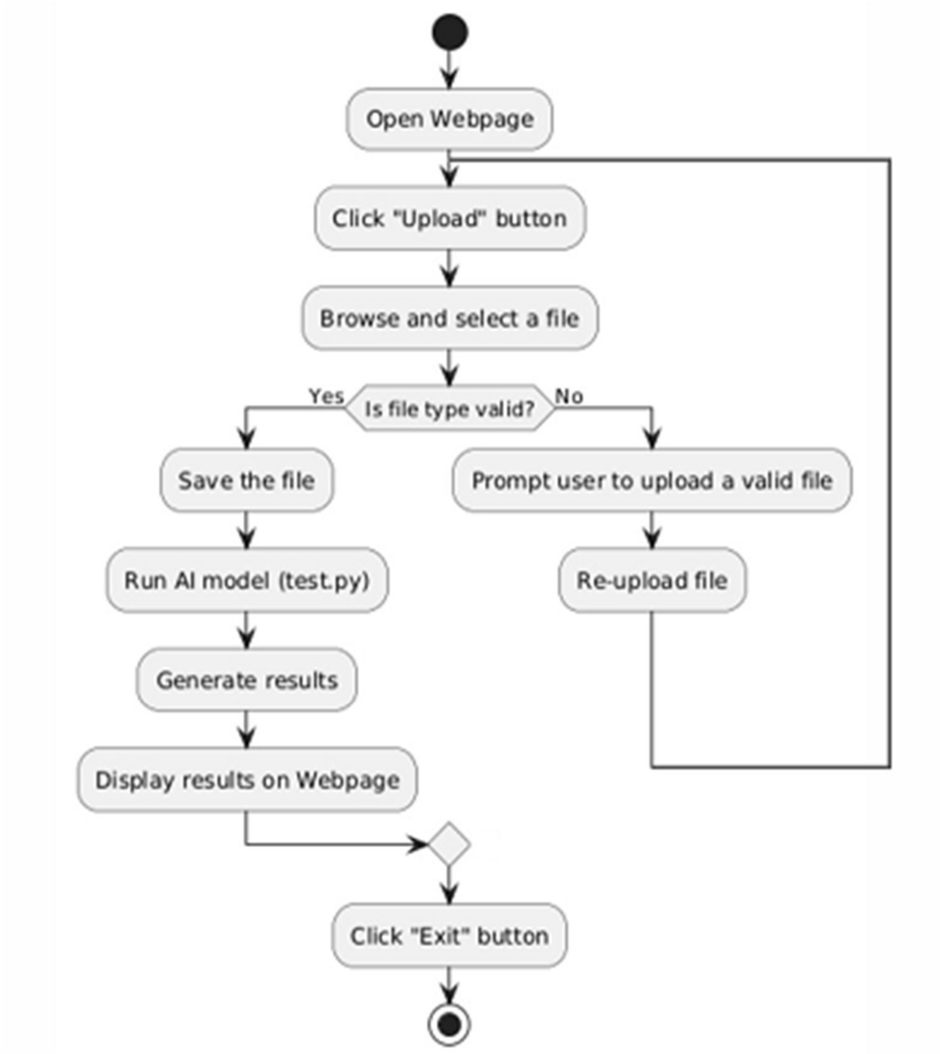


3.2. Class Diagram

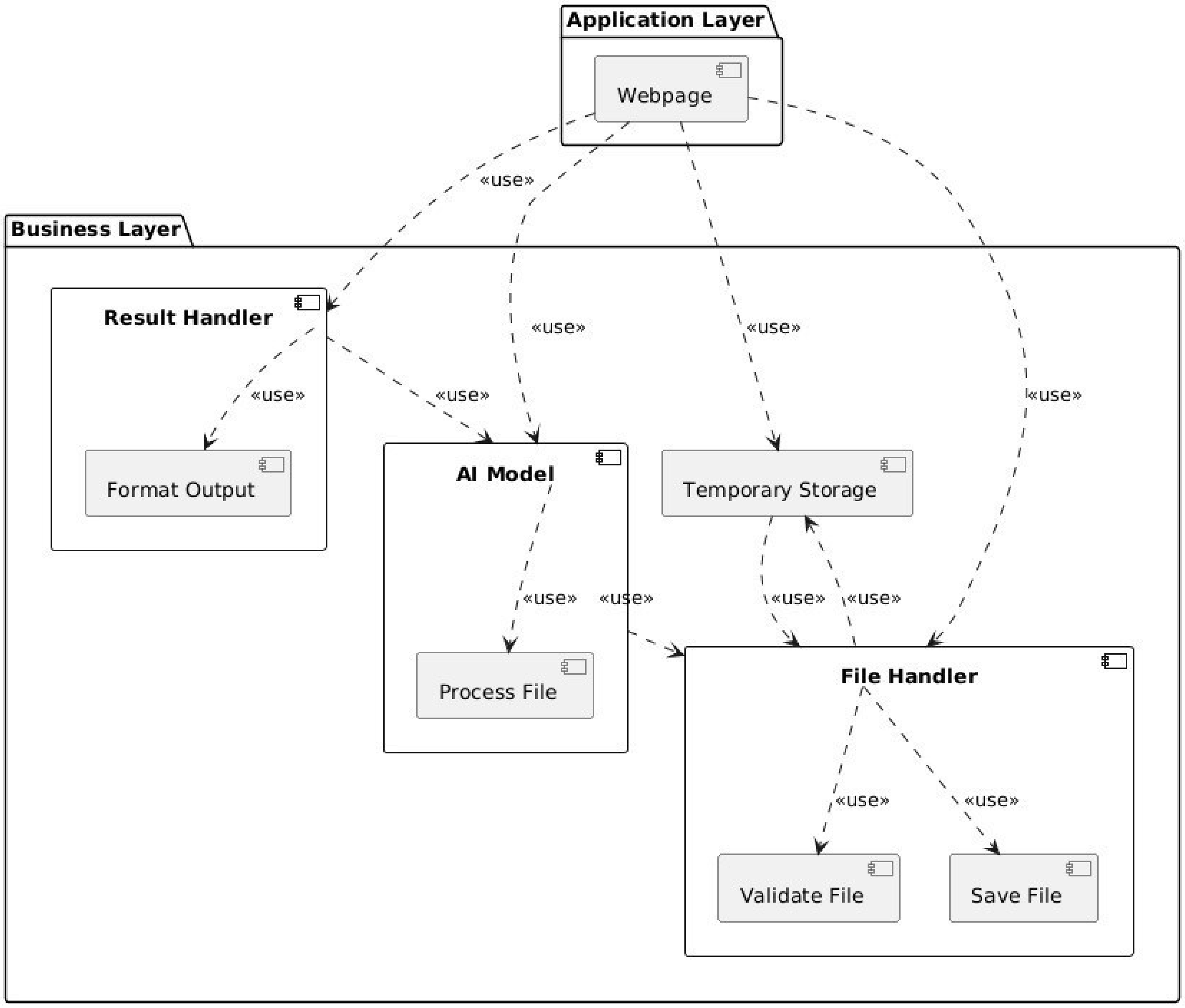
Class Diagram as shown below provides an overview of the target system by describing the objects and classes inside the system and the relationships between them. It provides a wide variety of usages; from modeling the domain-specific data structure to detailed design of the target system.



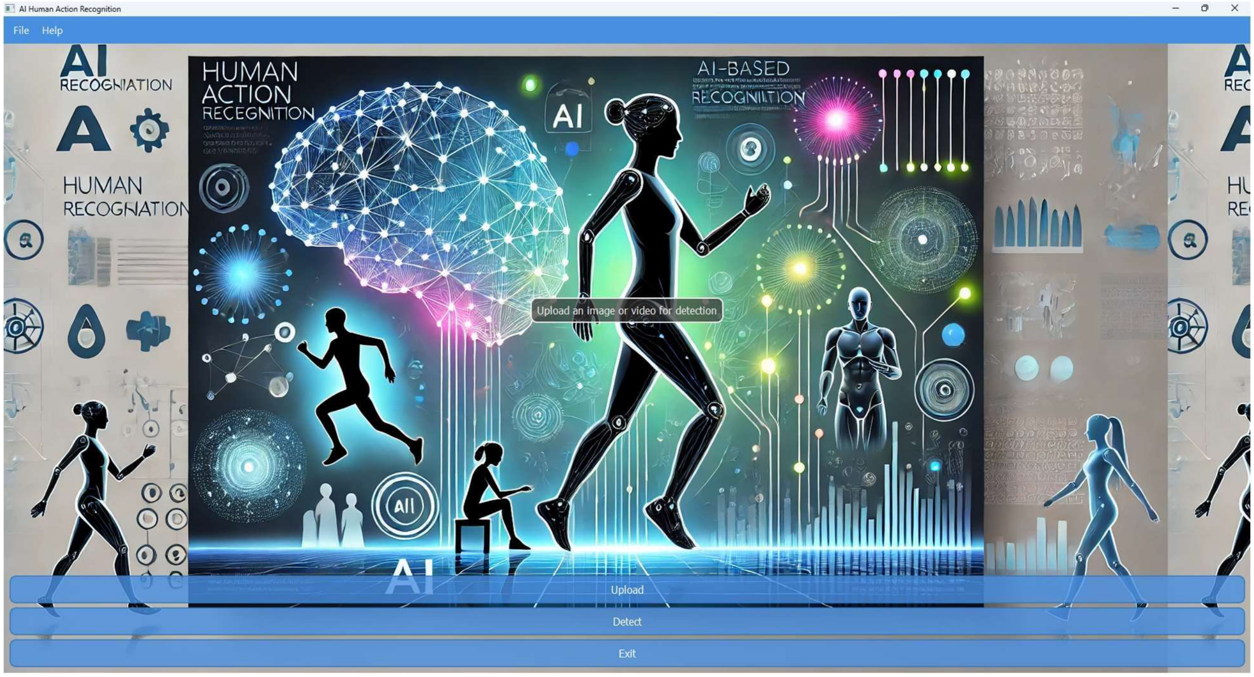
3.3. Activity Diagram



3.4. Architecture Diagram



3.4. Front-End Interface Design



Upload an image or video for detection

Chapter 4: Software Development

4.1. Software Design

Model Creation Code

```
import os
import cv2
import numpy as np
import tensorflow as tf
from keras import Sequential, layers
from keras.callbacks import EarlyStopping, LearningRateScheduler
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import ssl

# Bypass SSL Verification
ssl._create_default_https_context = ssl._create_unverified_context

# Directories for training and testing data
train_dir = "/Users/fawadnaveed/Desktop/DAA Assignment/train"
test_dir = "/Users/fawadnaveed/Desktop/DAA Assignment/test"

# Parameters
IMG_SIZE = 128  # Image size (128x128 pixels)
BATCH_SIZE = 32
EPOCHS = 50
NUM_CLASSES = 15

# Data Augmentation for training
train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    rescale=1.0 / 255.0,
    rotation_range=30,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    brightness_range=[0.8, 1.2],
    fill_mode='nearest'
)

# Data preprocessing for testing
test_datagen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1.0 / 255.0)

# Load data
```

```
train_data = train_datagen.flow_from_directory(
    train_dir,
    target_size=(IMG_SIZE, IMG_SIZE),
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

test_data = test_datagen.flow_from_directory(
    test_dir,
    target_size=(IMG_SIZE, IMG_SIZE),
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    shuffle=False
)

print(f"Classes in training data: {train_data.class_indices}")
print(f"Classes in testing data: {test_data.class_indices}")

# Use MobileNetV2 as the base model
base_model = tf.keras.applications.MobileNetV2(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=False,
    weights='imagenet'
)

# Freeze most of the layers
for layer in base_model.layers[:-30]:
    layer.trainable = False

# Build the model
model = Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.BatchNormalization(),
    layers.Dense(128, activation='swish', kernel_regularizer=tf.keras.regularizers.l2(1e-4)),
    layers.Dropout(0.4),
    layers.BatchNormalization(),
    layers.Dense(64, activation='swish', kernel_regularizer=tf.keras.regularizers.l2(1e-4)),
    layers.Dropout(0.3),
    layers.Dense(NUM_CLASSES, activation='softmax')
])

# Compile the model with label smoothing
model.compile(
    optimizer=tf.keras.optimizers.AdamW(learning_rate=1e-4, weight_decay=1e-4),
    loss=tf.keras.losses.CategoricalCrossentropy(label_smoothing=0.1),
    metrics=['accuracy']
)
```

```python
# Define callbacks
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

lr_schedule = tf.keras.optimizers.schedules.CosineDecay(
    initial_learning_rate=1e-4,
    decay_steps=10000,
    alpha=1e-6
)

# Train the model
history = model.fit(
    train_data,
    epochs=EPOCHS,
    validation_data=test_data,
    callbacks=[early_stopping]
)

# Save the model
model.save("human_action_model_improved_again.h5")

# Evaluate the model
test_loss, test_acc = model.evaluate(test_data)
print(f"Test Accuracy: {test_acc * 100:.2f}%")

# Classification Report
y_true = test_data.classes
y_pred = np.argmax(model.predict(test_data), axis=1)

print("\nClassification Report:\n")
print(classification_report(y_true, y_pred, target_names=list(test_data.class_indices.keys())))

# Confusion Matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(10, 8))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.colorbar()
tick_marks = np.arange(NUM_CLASSES)
plt.xticks(tick_marks, list(test_data.class_indices.keys()), rotation=45)
plt.yticks(tick_marks, list(test_data.class_indices.keys()))
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()

# Test with an example image
def predict_action(image_path, model, class_indices):
    image = cv2.imread(image_path)
    image = cv2.resize(image, (IMG_SIZE, IMG_SIZE))
    image = np.expand_dims(image, axis=0) / 255.0
    prediction = model.predict(image)
    predicted_class = list(class_indices.keys())[np.argmax(prediction)]
    return predicted_class

example_image_path = "/Users/fawadnaveed/Desktop/SEProject/Structured/test/calling/Image_10899.jpg"
predicted_action = predict_action(example_image_path, model, test_data.class_indices)
print(f"The predicted action is: {predicted_action}")
```

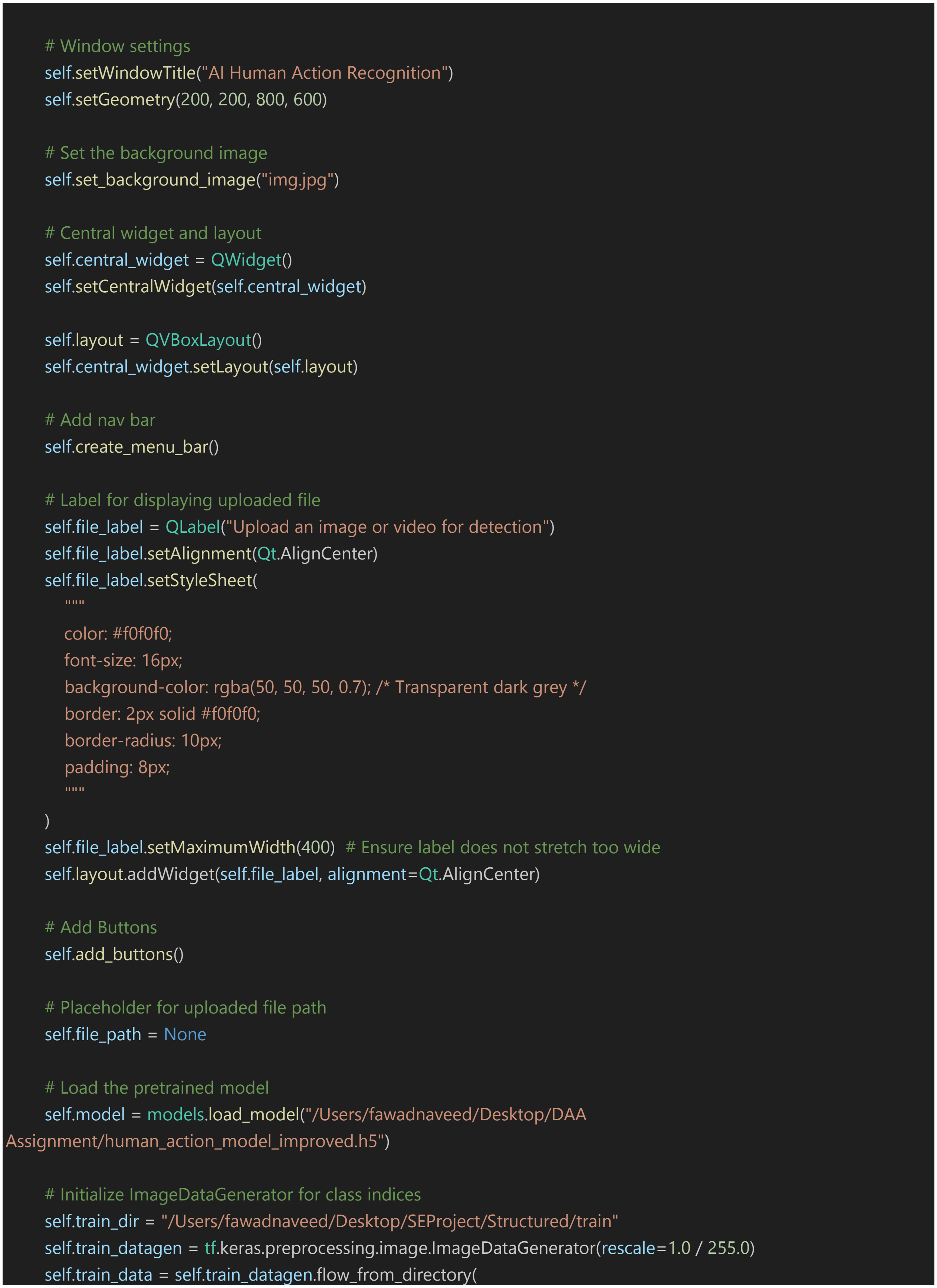
Model Accuracy Testing Code

```
from collections import Counter
import os
import cv2
import numpy as np
import keras
import tensorflow as tf
from keras import models
import matplotlib.pyplot as plt

IMG_SIZE = 128

# Load the saved model
model = models.load_model("/Users/fawadnaveed/Desktop/DAA Assignment/human_action_model_improved.h5")  #
Update this with the correct path to your saved model

# Directories for training and testing data
train_dir = "/Users/fawadnaveed/Desktop/SEProject/Structured/train"
test_dir = "/Users/fawadnaveed/Desktop/SEProject/Structured/test"

# Data generators
train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1.0 / 255.0)
test_datagen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1.0 / 255.0)

# Load training data
train_data = train_datagen.flow_from_directory(
    train_dir,
    target_size=(IMG_SIZE, IMG_SIZE),
    batch_size=32,
    class_mode='sparse'
)
model.compile(
    loss="sparse_categorical_crossentropy",
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    metrics=["accuracy"]  # Ensure a flat list of metric objects
)
```

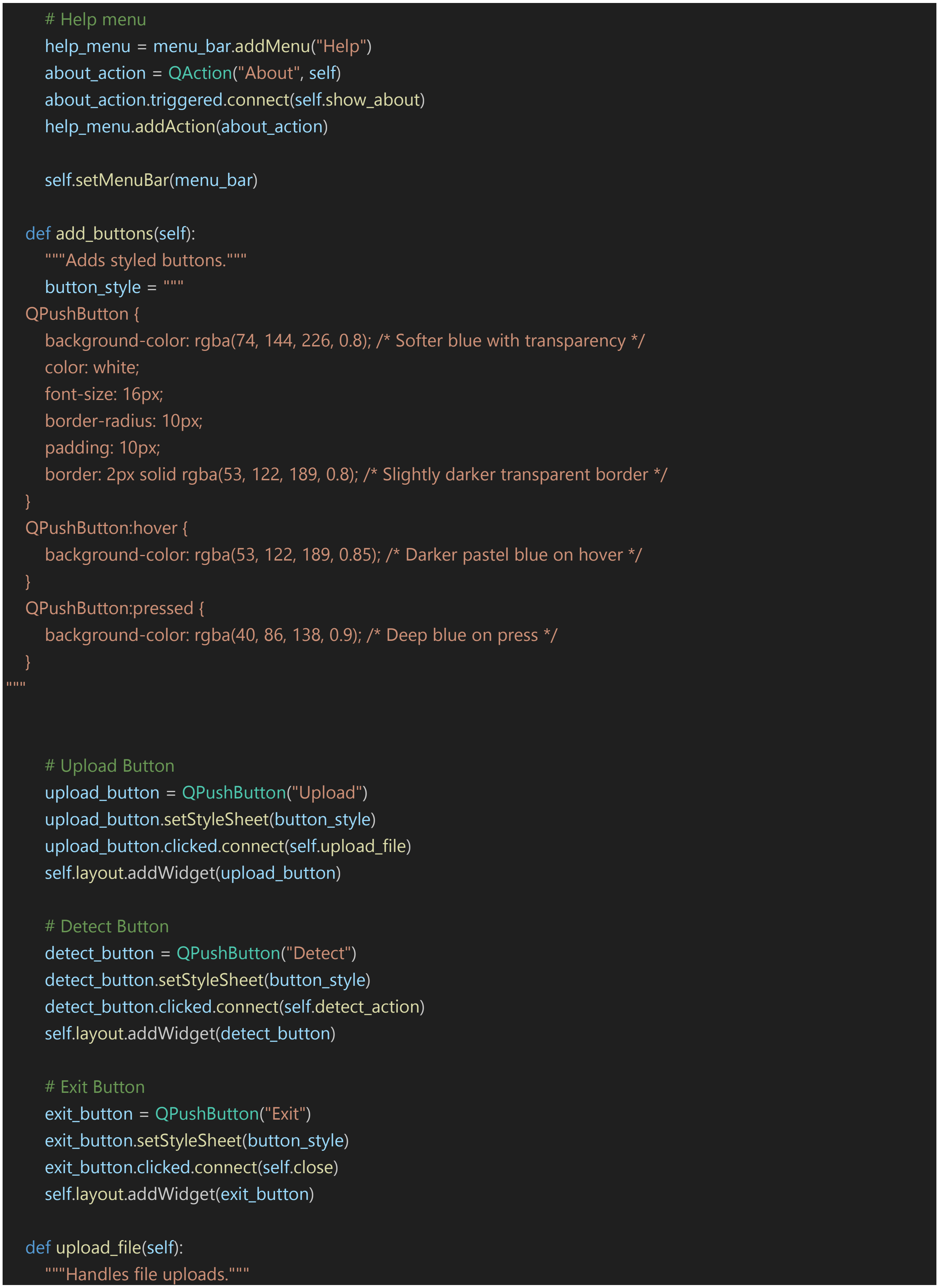
```
# Print class distribution in training data
print("Class distribution in training data:")
print(Counter(train_data.classes))

# Load testing data
test_data = test_datagen.flow_from_directory(
    test_dir,
    target_size=(IMG_SIZE, IMG_SIZE),
    batch_size=32,
    class_mode='sparse',
    shuffle=False  # Ensure consistent evaluation order
)

# Evaluate the model on the test dataset
print("\nEvaluating the model on test data...")

# Display model accuracy
test_loss, test_acc = model.evaluate(test_data)
print(f"Test Accuracy: {test_acc * 100:.2f}%")
```

Front End Code

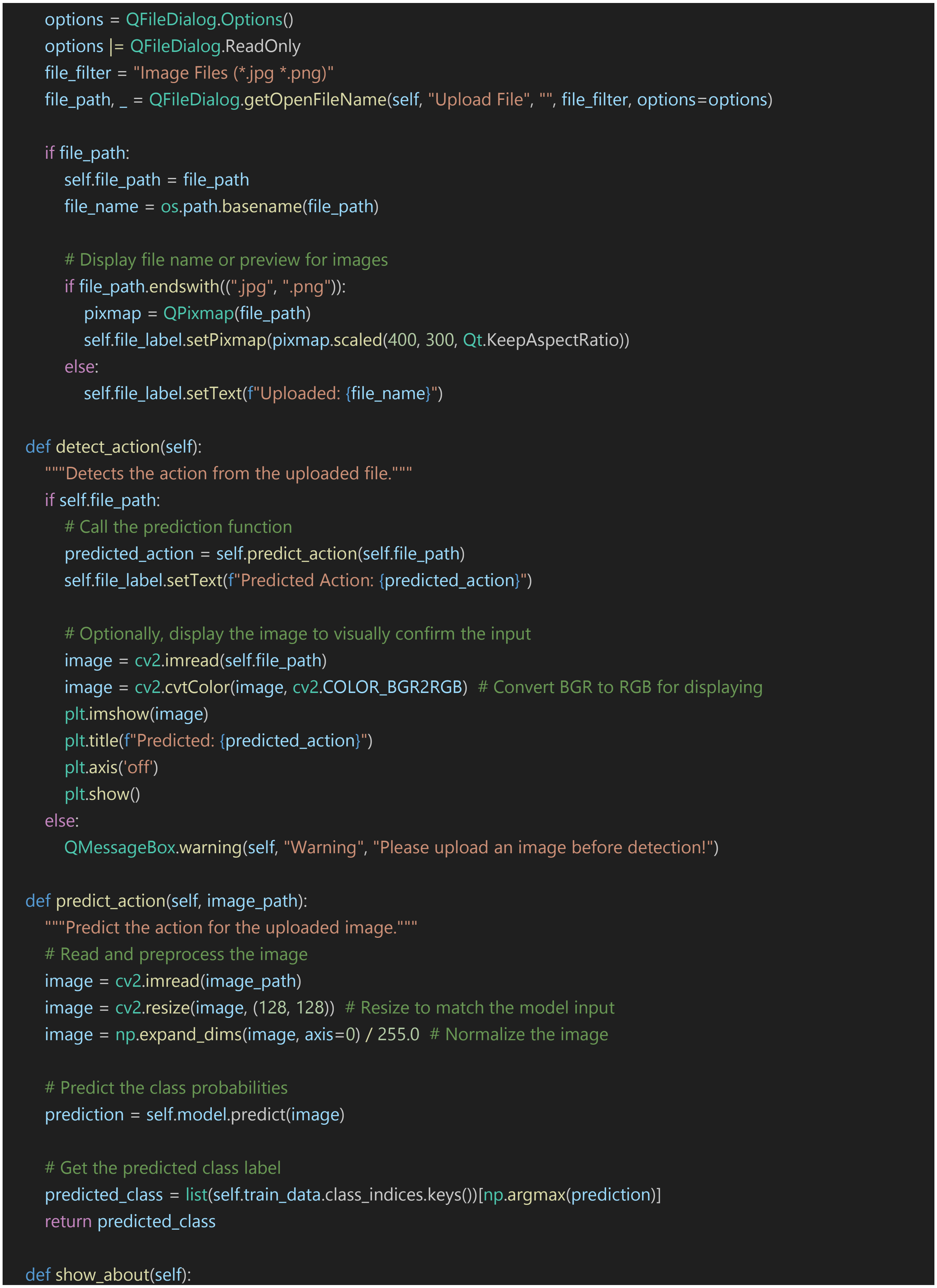
```
import sys
import os
import cv2
import numpy as np
from PyQt5.QtWidgets import (
    QApplication,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
    QFileDialog,
    QLabel,
    QMenuBar,
    QAction,
    QMessageBox,
)
from PyQt5.QtGui import QPixmap, QPalette, QBrush
from PyQt5.QtCore import Qt
import tensorflow as tf
from keras import models
import matplotlib.pyplot as plt

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
```



```
        # Window settings
        self.setWindowTitle("AI Human Action Recognition")
        self.setGeometry(200, 200, 800, 600)

        # Set the background image
        self.set_background_image("img.jpg")

        # Central widget and layout
        self.central_widget = QWidget()
        self.setCentralWidget(self.central_widget)

        self.layout = QVBoxLayout()
        self.central_widget.setLayout(self.layout)

        # Add nav bar
        self.create_menu_bar()

        # Label for displaying uploaded file
        self.file_label = QLabel("Upload an image or video for detection")
        self.file_label.setAlignment(Qt.AlignCenter)
        self.file_label.setStyleSheet(
            """
            color: #f0f0f0;
            font-size: 16px;
            background-color: rgba(50, 50, 50, 0.7); /* Transparent dark grey */
            border: 2px solid #f0f0f0;
            border-radius: 10px;
            padding: 8px;
            """
        )
        self.file_label.setMaximumWidth(400)  # Ensure label does not stretch too wide
        self.layout.addWidget(self.file_label, alignment=Qt.AlignCenter)

        # Add Buttons
        self.add_buttons()

        # Placeholder for uploaded file path
        self.file_path = None

        # Load the pretrained model
        self.model = models.load_model("/Users/fawadnaveed/Desktop/DAA
Assignment/human_action_model_improved.h5")

        # Initialize ImageDataGenerator for class indices
        self.train_dir = "/Users/fawadnaveed/Desktop/SEProject/Structured/train"
        self.train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1.0 / 255.0)
        self.train_data = self.train_datagen.flow_from_directory(
            self.train_dir,
            target_size=(128, 128),
            batch_size=32,
            class_mode='categorical'
        )

    def set_background_image(self, image_path):
        """Sets a background image for the main window."""
        palette = QPalette()
        pixmap = QPixmap(image_path)
        palette.setBrush(QPalette.Background, QBrush(pixmap))
        self.setPalette(palette)

    def create_menu_bar(self):
        """Creates a styled navigation bar."""
        menu_bar = QMenuBar(self)
        menu_bar.setStyleSheet(
            """
            QMenuBar {
                background-color: #4a90e2; /* Soft pastel blue */
                color: white;
                font-size: 16px;
                padding: 5px;
            }
            QMenuBar::item {
                background-color: transparent;
                padding: 5px 10px;
            }
            QMenuBar::item:selected {
                background-color: #357abd; /* Slightly darker pastel blue */
            }
            QMenu {
                background-color: #4a90e2;
                color: white;
                margin: 2px;
            }
            QMenu::item:selected {
                background-color: #357abd;
            }
            """
        )

        # File menu
        file_menu = menu_bar.addMenu("File")
        exit_action = QAction("Exit", self)
        exit_action.triggered.connect(self.close)
        file_menu.addAction(exit_action)
```



```
        # Help menu
        help_menu = menu_bar.addMenu("Help")
        about_action = QAction("About", self)
        about_action.triggered.connect(self.show_about)
        help_menu.addAction(about_action)

        self.setMenuBar(menu_bar)

    def add_buttons(self):
        """Adds styled buttons."""
        button_style = """
    QPushButton {
        background-color: rgba(74, 144, 226, 0.8); /* Softer blue with transparency */
        color: white;
        font-size: 16px;
        border-radius: 10px;
        padding: 10px;
        border: 2px solid rgba(53, 122, 189, 0.8); /* Slightly darker transparent border */
    }
    QPushButton:hover {
        background-color: rgba(53, 122, 189, 0.85); /* Darker pastel blue on hover */
    }
    QPushButton:pressed {
        background-color: rgba(40, 86, 138, 0.9); /* Deep blue on press */
    }
"""


        # Upload Button
        upload_button = QPushButton("Upload")
        upload_button.setStyleSheet(button_style)
        upload_button.clicked.connect(self.upload_file)
        self.layout.addWidget(upload_button)

        # Detect Button
        detect_button = QPushButton("Detect")
        detect_button.setStyleSheet(button_style)
        detect_button.clicked.connect(self.detect_action)
        self.layout.addWidget(detect_button)

        # Exit Button
        exit_button = QPushButton("Exit")
        exit_button.setStyleSheet(button_style)
        exit_button.clicked.connect(self.close)
        self.layout.addWidget(exit_button)

    def upload_file(self):
        """Handles file uploads."""
```



```
    options = QFileDialog.Options()
    options |= QFileDialog.ReadOnly
    file_filter = "Image Files (*.jpg *.png)"
    file_path, _ = QFileDialog.getOpenFileName(self, "Upload File", "", file_filter, options=options)

    if file_path:
        self.file_path = file_path
        file_name = os.path.basename(file_path)

        # Display file name or preview for images
        if file_path.endswith((".jpg", ".png")):
            pixmap = QPixmap(file_path)
            self.file_label.setPixmap(pixmap.scaled(400, 300, Qt.KeepAspectRatio))
        else:
            self.file_label.setText(f"Uploaded: {file_name}")

def detect_action(self):
    """Detects the action from the uploaded file."""
    if self.file_path:
        # Call the prediction function
        predicted_action = self.predict_action(self.file_path)
        self.file_label.setText(f"Predicted Action: {predicted_action}")

        # Optionally, display the image to visually confirm the input
        image = cv2.imread(self.file_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Convert BGR to RGB for displaying
        plt.imshow(image)
        plt.title(f"Predicted: {predicted_action}")
        plt.axis('off')
        plt.show()
    else:
        QMessageBox.warning(self, "Warning", "Please upload an image before detection!")

def predict_action(self, image_path):
    """Predict the action for the uploaded image."""
    # Read and preprocess the image
    image = cv2.imread(image_path)
    image = cv2.resize(image, (128, 128))  # Resize to match the model input
    image = np.expand_dims(image, axis=0) / 255.0  # Normalize the image

    # Predict the class probabilities
    prediction = self.model.predict(image)

    # Get the predicted class label
    predicted_class = list(self.train_data.class_indices.keys())[np.argmax(prediction)]
    return predicted_class

def show_about(self):
        """Shows an About dialog."""
        QMessageBox.information(
            self,
            "About",
            "This is a PyQt-based GUI for an AI Human Action Recognition model.\n\n"
            "1. Upload: Uploads an image or video.\n"
            "2. Detect: Runs the AI model.\n"
            "3. Exit: Closes the application.",
        )


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.showMaximized()
    sys.exit(app.exec_())
```

Chapter 5: Software Testing

5.1. Test Plan

5.1.1. Testing Strategy

- **Unit Testing**: Validate individual components (e.g., video preprocessing, model inference).
- **Integration Testing**: Ensure smooth interaction between backend AI models and frontend.
- **System Testing**: Verify the entire system functionality.
- **Performance Testing**: Test response times for real-time recognition.

5.1.2 Types of Tests

1. **Functional Tests**:
   - Test action classification accuracy.
   - Test user interface components (upload, detect, and help).
2. **Performance Tests**:
   - Measure response time for live action detection.
3. **Usability Tests**:
   - Evaluate user interface ease of use for non-technical users.

5.1.3 Test Cases

- **Case 1**: Verify action classification for uploaded picture.
  - Input: Pictures of various actions.
  - Expected Output: Correct classification of action inside the pictures.
  -

5.2. Test Report

Summary of Testing Activities

- Total Test Cases Executed: 40
- Passed: 32
- Failed: 7

Defects Found

1. Inconsistent action classification for overlapping actions in the feeds.

Resolutions

- Improved preprocessing pipeline to handle overlapping actions.

Chapter 6: Software Deployment

6.1 Steps for Deployment

1. **Prepare Environment**:
   - Set up servers with required libraries (TensorFlow, OpenCV).
   - Configure hardware (cameras, GPUs).
   - Install dependencies such as Python, NumPy, and Keras.
2. **Deploy Application**:
   - Upload trained model and codebase to the server.
   - Set up database for user interactions and results storage.
3. **Testing**:
   - Conduct final system testing in live environments.
   - Validate real-time performance and user interface responsiveness.
4. **Launch**:
   - Make the application available to end-users via web or mobile platforms.
   - Provide documentation and a support contact for users.

6.2. User Manual

Instructions

1. **Uploading Media**:
    - Click the "Upload" button to select video or image files.
    - Ensure file type is supported (e.g., MP4, JPG).
2. **Running Detection**:
    - Click "Detect" after uploading media.
    - View results on the interface.

Troubleshooting

- **Issue**: Detection fails to start.
    - **Solution**: Verify that the uploaded file meets format and resolution requirements.
- **Issue**: Detection takes too much time.
    - **Solution**: End the backend processes consuming RAM and GPU.

6.2. Maintenance Plan

Post-Deployment Guidelines

1. Regularly update the AI model with new data for improved accuracy.
    - Schedule periodic dataset enhancements and retraining.
2. Monitor server performance and optimize resources.
    - Use monitoring tools to track CPU/GPU utilization and latency.
3. Provide periodic software updates for bug fixes and new features.
    - Maintain a changelog for transparency.
4. Maintain user documentation to reflect system updates.
    - Ensure instructions remain relevant and comprehensive.

Support and Feedback

- Establish a dedicated support team for handling user queries.
- Set up a feedback form for collecting user input on functionality and usability.
- Implement a ticketing system for tracking and resolving issues promptly.