Q1

1. Data Loading and Preprocessing

The temperature dataset was loaded from an external file using the Pandas library. The file contains time-stamped environmental measurements, including indoor and outdoor temperature, humidity, CO_2 levels, lighting, and meteorological variables. After loading the dataset, meaningful column names were assigned to improve readability and data handling. The target variable was selected as the indoor bedroom temperature (Temperature_Habitacion), which was subsampled at regular intervals to reduce temporal redundancy and computational complexity.

2. Time-Series Feature Engineering

To model temporal dependencies in the temperature data, a sliding window approach was applied. The prediction task was formulated as a univariate time-series regression problem, where the next temperature value is predicted using the previous five observed values. These lagged temperature values were constructed as input features (t-1 to t-5) by shifting the target series accordingly. Rows containing undefined values resulting from the shifting operation were removed to ensure consistency between input features and target labels.

3. Dataset Splitting

The constructed dataset was divided into training and testing subsets using an 80/20 split. A fixed random seed was used during the split to ensure reproducibility of the results. This separation allows the model to be trained on historical data while being evaluated on unseen samples to assess generalization performance.

4. Linear Regression Model Training

A linear regression model was selected due to its simplicity and suitability for modeling linear temporal relationships. The model was trained using the training subset, learning the relationship between past temperature values and the future temperature prediction. Once training was completed, predictions were generated for both the training and test datasets.

5. Model Evaluation and Visualization

Model performance was evaluated using the mean absolute error (MAE) metric for both training and test sets. To improve interpretability, the square root of the MAE was reported. Additionally, a visualization was generated comparing the predicted temperature values against the actual test values, allowing a qualitative assessment of the model's prediction accuracy and temporal alignment.

6. Model Persistence

After training and evaluation, the trained linear regression model was serialized and saved to disk using the Joblib library. Storing the model enables reuse in future experiments or deployment without the need to retrain, supporting modularity and reproducibility within the overall system.

7. Summary

At the end of this stage, a time-series–based linear regression model was successfully trained to predict indoor temperature using historical measurements. The implementation demonstrates a complete machine learning workflow, including data preprocessing, feature engineering, model training, evaluation, visualization, and persistence, forming a reliable foundation for further optimization or deployment.

Q2

1. Embedded System Setup

After completing the model training and quantization steps in the Python environment, the trained human activity recognition model was deployed on an STM32 microcontroller using Mbed Studio.
A bare-metal configuration was selected instead of an RTOS-based setup to minimize memory overhead and ensure deterministic execution, which is critical for lightweight machine learning inference on resource-constrained devices.

The project was initialized using the *Blinky Baremetal* template and configured for the target STM32 board.

2. TensorFlow Lite Micro Integration

To enable on-device inference, **TensorFlow Lite Micro (TFLM)** was integrated into the Mbed project. Unlike standard TensorFlow, TFLM is specifically designed for microcontrollers and does not require an operating system.

Only the essential components of the TensorFlow Lite Micro framework were included in the project to reduce memory usage:

- tensorflow/lite/micro – Micro interpreter and runtime
- tensorflow/lite/kernels – Required kernel implementations

- tensorflow/lite/schema – Model schema definitions

All desktop-oriented components (such as compiler, MLIR, examples, tools, and benchmarks) were excluded from the build, as they are not supported on microcontroller platforms.

3. Model Deployment

The trained and quantized INT8 TensorFlow Lite model was converted into a C-compatible array format (har_model_data.cc).
This file contains the serialized model binary and is directly compiled into the firmware, eliminating the need for file systems or external storage on the microcontroller.

The model was accessed in the application code using external references:

```
extern const unsigned char g_har_model[];
extern const unsigned int g_har_model_len;
```

4. Operator Resolver Configuration

Since TensorFlow Lite Micro does not automatically include all operators, the required operations must be explicitly registered.
The deployed model consists of a single fully connected layer with sigmoid activation, therefore only the FullyConnected operator was registered.

```
static tflite::MicroMutableOpResolver<1> resolver;
resolver.AddFullyConnected();
```

This approach significantly reduces flash and RAM usage compared to registering all available operators and is well suited for embedded machine learning applications.

5. Memory Management

A static memory region called the **tensor arena** was allocated to store all intermediate tensors required during inference. The size of the tensor arena was carefully chosen to balance memory constraints and model requirements.

```
constexpr int kTensorArenaSize = 20 * 1024;
alignas(16) static uint8_t tensor_arena[kTensorArenaSize];
```

Static memory allocation ensures predictable memory usage and avoids dynamic memory fragmentation, which is critical in embedded systems.

6. Inference Pipeline

The TensorFlow Lite Micro interpreter was initialized using the deployed model, operator resolver, and tensor arena. Input features were manually quantized from floating-point values to INT8 format using the scale and zero-point parameters provided by the model.

After invoking the interpreter, the output was dequantized back to floating-point format and interpreted as a probability value. A threshold of 0.5 was used to classify the activity:

- Output ≤ 0.5 → Walking
- Output > 0.5 → Not Walking

The classification result was transmitted via the serial interface for debugging and verification.

7. Summary

At the end of this stage, the human activity recognition model was successfully deployed and executed on an STM32 microcontroller using TensorFlow Lite Micro. The implementation demonstrates that a lightweight, single-neuron neural network can be efficiently executed on resource-constrained embedded hardware while maintaining correct inference behavior.

Q3

1. Embedded System Setup

After completing the training of the keyword spotting model in the Python environment, the trained single-neuron speech recognition model was deployed on an STM32 microcontroller using Mbed Studio. The application was designed to distinguish between two classes: “zero” and “not zero”.

A bare-metal configuration was selected instead of an RTOS-based setup in order to minimize memory overhead and ensure deterministic execution. This design choice is particularly important for lightweight neural network inference on resource-constrained microcontroller platforms.

The project was initialized using the Blinky Baremetal template and configured for the target STM32 development board. This setup provided a minimal runtime environment suitable for integrating TensorFlow Lite Micro.

2. TensorFlow Lite Micro Integration

To enable on-device inference, TensorFlow Lite Micro (TFLM) was integrated into the Mbed project. Unlike standard TensorFlow, TFLM is specifically optimized for microcontroller-based systems and does not rely on an operating system or dynamic memory allocation.

To reduce both flash and RAM usage, only the essential TensorFlow Lite Micro components were included:

- tensorflow/lite/micro – Micro interpreter and runtime
- tensorflow/lite/kernels – Required kernel implementations
- tensorflow/lite/schema – Model schema definitions

All desktop-oriented components such as MLIR, converters, examples, tools, and benchmarks were excluded from the build, as they are not supported or required on embedded platforms.

3. Model Deployment

The trained TensorFlow Lite model was converted into a C-compatible byte array and embedded directly into the firmware as kws_perceptron_model_data.cc. This file contains the serialized .tflite model and allows the neural network to be compiled directly into the application binary.

This approach eliminates the need for external storage or file systems on the microcontroller.

The model was accessed in the application code using external references:

```
extern const unsigned char kws_perceptron_tflite[];
extern const unsigned int kws_perceptron_tflite_len;
```

The model integrity was verified at runtime by checking the TensorFlow Lite schema version.

4. Operator Resolver Configuration

TensorFlow Lite Micro does not automatically include all neural network operators. Instead, only the operators required by the deployed model must be explicitly registered.

The keyword spotting model consists of a single fully connected layer with sigmoid activation. Therefore, only the FullyConnected operator was registered in the mutable operator resolver:

```
static tflite::MicroMutableOpResolver<1> resolver;
resolver.AddFullyConnected();
```

By registering only the necessary operator, the memory footprint of the application was significantly reduced, making the solution well suited for embedded keyword spotting applications.

5. Memory Management

All intermediate tensors required during inference were stored in a statically allocated memory region known as the tensor arena. The tensor arena size was selected based on the model requirements and available SRAM on the target device.

```
constexpr int kTensorArenaSize = 8 * 1024;
alignas(16) static uint8_t tensor_arena[kTensorArenaSize];
```

Static memory allocation ensures predictable memory usage and avoids heap fragmentation, which is critical for reliable execution in embedded systems.

6. Inference Pipeline

After initializing the TensorFlow Lite Micro interpreter with the deployed model, operator resolver, and tensor arena, the input tensor was obtained from the interpreter. The input vector represents the MFCC-based audio features used during training. For testing purposes, the input tensor was populated with example values, while in a real deployment scenario these values would be computed from incoming audio samples.

The interpreter was then invoked to perform inference:

```
interpreter.Invoke();
```

The output tensor contains a single value representing the probability of the keyword “zero”. The output was interpreted as follows:

- Output ≤ 0.5 → Not Zero
- Output > 0.5 → Zero

The resulting prediction value was transmitted via the serial interface for debugging and validation purposes.

7. Summary

At the end of this stage, the keyword spotting model was successfully deployed and executed on an STM32 microcontroller using TensorFlow Lite Micro. The implementation demonstrates that a single-neuron neural network can perform speech classification efficiently on resource-constrained embedded hardware. This deployment confirms that lightweight neural network architectures combined with TensorFlow Lite Micro provide a practical and memory-efficient solution for real-time embedded keyword spotting applications.

Q4

1. Embedded System Setup

After training and converting the HDR perceptron model to TensorFlow Lite format, the model was deployed on an STM32 microcontroller using Mbed Studio. A bare-metal configuration was selected to minimize runtime overhead and ensure deterministic execution suitable for embedded machine learning inference. The application was implemented as a standalone firmware project targeting the STM32 platform, without reliance on an RTOS, enabling efficient use of limited computational and memory resources.

2. TensorFlow Lite Micro Integration

To enable on-device inference, TensorFlow Lite Micro (TFLM) was integrated into the embedded project. TFLM is designed specifically for microcontrollers and operates without an operating system, making it well suited for this deployment scenario. Only the essential TFLM components required for inference were included, such as the micro interpreter, operator resolver, and schema definitions. Desktop-oriented TensorFlow components were excluded to reduce flash and RAM usage and maintain compatibility with the embedded environment.

3. Model Deployment

The trained perceptron model was converted into a C-compatible byte array (hdr_perceptron_model) and compiled directly into the firmware. This approach embeds the serialized TensorFlow Lite model binary into program memory, eliminating the need for file systems or external storage. The model is accessed through external references to the model data array and its length, allowing the interpreter to load the model directly from flash memory at runtime.

4. Operator Resolver Configuration

TensorFlow Lite Micro requires explicit registration of all operators used by the model. Since the deployed model consists of a single fully connected layer followed by a sigmoid activation function, only the necessary operators were registered. A MicroMutableOpResolver was configured to include the FullyConnected and Logistic operators. This selective registration significantly reduces memory consumption compared to including the full set of available operators and is appropriate for lightweight neural network inference on embedded devices.

5. Memory Management

A static memory region known as the tensor arena was allocated to store all tensors required during inference, including input, output, and intermediate buffers. The tensor arena size was set to 4 KB based on the model's memory requirements. Static allocation ensures predictable memory usage and avoids dynamic memory fragmentation, which is critical for reliability and stability in embedded systems.

6. Inference Pipeline

The TensorFlow Lite Micro interpreter was initialized using the deployed model, operator resolver, and tensor arena. The input tensor was populated with seven floating-point Hu moment values, corresponding to the model's expected input feature vector. After invoking the interpreter, the output tensor produced a single floating-point value representing the sigmoid-activated perceptron output. This value corresponds to the model's prediction and was printed via the serial interface for debugging and validation purposes.

7. Summary

At the end of this stage, the HDR perceptron model was successfully deployed and executed on an STM32 microcontroller using TensorFlow Lite Micro. The implementation demonstrates that a simple single-layer neural network with sigmoid activation can be efficiently executed on resource-constrained embedded hardware. The system achieves reliable inference using minimal memory and computational resources, validating the feasibility of deploying lightweight machine learning models in embedded environments.