HOMEWORK-2 REPORT

CSE 421

Efe Kayadelen
Student ID : 20220701088

Samet Alper Özdemir
Student ID : 20210702021

GITHUB REPO: https://github.com/blackwh1p/CSE421-Homeworks

Part-1:

The Python code acts as the host; it trains the Naive Bayes model, converts it into C code for efficient deployment, and manages the entire testing process—sending prepared test data to the hardware and collecting the results. The C code operates on the target microcontroller; it uses the pre-trained C model parameters to execute real-time feature extraction and classification on the incoming sensor data.

Together, the scripts enable accurate performance measurement of the classified activity by comparing the microcontroller's real-time predictions against the known true labels on the Python host.

Our Codes:

- C Code:

```c
#include "mbed.h"
#include "arm_math.h"             // Includes the CMSIS-DSP library for efficient math operations
#include "har_feature_extraction.h"
#include "bayes_cls_inference.h"
#include "bayes_cls_config.h"      // Contains the parameters of the trained Bayes Classifier (exported from Python)

#define BAUD_RATE 115200
#define WINDOW_SIZE 64             // Matches the segmentation window used in the Python training script
#define NUM_AXES 3

UnbufferedSerial pc(USBTX, USBRX); // Setup serial communication over USB (for data transfer to/from host PC)

// Buffers
float32_t acc_data[3][WINDOW_SIZE];  // Buffer to hold 3 axes (X, Y, Z) of accelerometer data, 64 samples each
HAR_FtrExtOutput feature_out;
float32_t feature_vector[NUM_FEATURES]; // The 10 features extracted from acc_data, used as model input
float32_t class_probs[NUM_CLASSES];     // Array to store the probability output for each activity class

// Matrix Wrappers
arm_matrix_instance_f32 mat_input;  // Wrapper structure for the input feature vector (required by CMSIS-DSP)
arm_matrix_instance_f32 mat_output; // Wrapper structure for the output class probabilities

int main() {
    pc.baud(BAUD_RATE);

    // --- FIX: Matrix Dimensions for Book Library ---
    // The library computes: Transpose(Input) * Covariance
    // Input (10x1) -> Transposed to (1x10)
    // (1x10) * (10x10) = (1x10) -> OK
    mat_input.numRows = NUM_FEATURES; // Defines the input as a 10x1 column vector
    mat_input.numCols = 1;
    mat_input.pData = feature_vector;

    mat_output.numRows = 1;
    mat_output.numCols = NUM_CLASSES; // Defines the output as a 1xN row vector (N = number of classes)
    mat_output.pData = class_probs;

    while (true) {
        // 1. Sync
        char ready = 'R';
        pc.write(&ready, 1);       // Signals the host Python script that the MCU is ready to receive data

        // 2. Receive Data
        char* ptr = (char*)acc_data;
        int bytes_remaining = sizeof(acc_data);
        while (bytes_remaining > 0) {
            if (pc.readable()) {
                int n = pc.read(ptr, bytes_remaining); // Blocks until a full window of 3-axis data (3*64 floats) is received
                bytes_remaining -= n;
                ptr += n;
            }
        }

        // 3. Feature Extraction
        har_extract_features(acc_data, &feature_out); // Computes the 10 features (Mean, Std Dev, FFT, etc.) from the raw data

        // 4. Fill Feature Vector (ORDER MATTERS)
        // Python adds Mean columns first
        feature_vector[0] = feature_out.x_mean;
        feature_vector[1] = feature_out.y_mean;
        feature_vector[2] = feature_out.z_mean;

        // Then Positive Count columns
        feature_vector[3] = feature_out.x_pos;
        feature_vector[4] = feature_out.y_pos;
        feature_vector[5] = feature_out.z_pos;

        // Then FFT Std Dev columns
        feature_vector[6] = feature_out.fft_sd_x;
        feature_vector[7] = feature_out.fft_sd_y;
        feature_vector[8] = feature_out.fft_sd_z;

        // Finally SMA
        feature_vector[9] = feature_out.sma; // **Crucial:** The order of features must exactly match the Python training order

        // 5. Inference
        bayes_cls_predict(&mat_input, &mat_output); // Runs the Naive Bayes classification using the imported C parameters

        // 6. Argmax
        int best_class = 0;
        float max_val = class_probs[0];
        for(int i=1; i<NUM_CLASSES; i++) {
            if (class_probs[i] > max_val) {
                max_val = class_probs[i];
                best_class = i;
            }
        } // Finds the index (ID) of the class with the highest probability (Argmax)

        char result = (char)best_class;
        pc.write(&result, 1);        // Sends the classification result (1 byte) back to the Python host
    }
}
```

- Python Code:

```
import os.path as osp
from data_utils import read_data
from feature_utils import create_features
from sklearn import metrics
import sklearn2c                    # Key library for converting scikit-learn models into C code
from matplotlib import pyplot as plt

# --- CHANGE: Set to power of 2 for STM32 FFT compatibility ---
TIME_PERIODS = 64                   # Window size for segmentation, a power of 2 for FFT efficiency
STEP_DISTANCE = 32                  # Overlap distance (50% overlap for TIME_PERIODS=64)
# ------------------------------------------------------------

DATA_PATH = osp.join("WISDM_ar_v1.1", "WISDM_ar_v1.1_raw.txt")

# Read and Filter Data
data_df = read_data(DATA_PATH)
df_train = data_df[data_df["user"] <= 28] # Defines the training set using data from a subset of users
#df_test = data_df[data_df["user"] > 28]

# Extract Features
train_segments_df, train_labels = create_features(df_train, TIME_PERIODS, STEP_DISTANCE) # Segments data and calculates features
(e.g., mean, FFT, standard deviation)
#test_segments_df, test_labels = create_features(df_test, TIME_PERIODS, STEP_DISTANCE)

# Train Bayes Classifier
bayes = sklearn2c.BayesClassifier()      # Instantiates the classifier intended for C conversion
bayes.train(train_segments_df, train_labels) # Trains the Naive Bayes model on the extracted features

# Evaluate
#bayes_preds = bayes.predict(test_segments_df)
#conf_matrix = metrics.confusion_matrix(test_labels, bayes_preds)
#print(f"Accuracy: {metrics.accuracy_score(test_labels, bayes_preds)}")

# Export for Microcontroller
# This generates 'bayes_cls_config.h' and 'bayes_har_config.c'
bayes.export("../bayes_cls_config")      # Converts the trained model parameters into C code files
print("Model exported to bayes_cls_config.h/.c") # Confirms successful creation of files for the MCU
```

Part-2:

The Python script (Host PC) manages the testing lifecycle: it calculates the MFCC features for test audio files (ensuring the feature vector format matches the target), opens a serial connection, and runs an evaluation loop. In this loop, it sends the feature vector to the microcontroller as bytes and then receives the raw prediction results as bytes. Finally, it uses these results to calculate the overall classification accuracy. The C code (Embedded Target) acts as a low-latency inference engine: it runs on an mbed OS board and continuously waits for and receives the feature vector over serial. Upon receipt, it calls knn_cls_predict() to execute the pre-trained k-NN classification and immediately sends the resulting prediction array back to the host PC for analysis. This setup ensures that the model's performance metrics reflect its true behavior when deployed on the resource-constrained target hardware.

Our Codes:

- C Code:

```c
#include "mbed.h"

extern "C" {
#include "knn_cls_inference.h"   // provides knn_cls_predict(), the function that runs the k-NN model
#include "ks_feature_extraction.h" // only for nbDctOutputs definition (defines the number of base features, e.g., 13 MFCCs)
}

// Serial port to PC (ST-LINK VCP on most Nucleo/Disco boards)
BufferedSerial pc(USBTX, USBRX, 115200);   // Initializes serial communication (interface with the Python host)

// Features: same size as before (nbDctOutputs * 2)
float32_t ExtractedFeatures[nbDctOutputs * 2] = {0}; // Buffer to hold the incoming feature vector (e.g., 26 features)
int32_t   output[NUM_CLASSES] = {0};                // Array to store the k-NN prediction scores/votes for each class

int main()
{
    pc.set_blocking(true); // Ensures serial communication waits until all data is sent or received

    const uint32_t num_features = nbDctOutputs * 2;
    const uint32_t bytes_to_read = num_features * sizeof(float32_t); // Calculates total bytes needed for the float32 input vector
    const uint32_t bytes_to_write = NUM_CLASSES * sizeof(int32_t); // Calculates total bytes for the int32 output vector

    while (true) { // The main inference loop
        uint8_t *rx_ptr = reinterpret_cast<uint8_t*>(ExtractedFeatures);
        uint32_t received = 0;

        // ---- Receive feature vector from PC (float32 LE) ----
        while (received < bytes_to_read) {
            ssize_t n = pc.read(rx_ptr + received, bytes_to_read - received); // Reads incoming bytes into the feature buffer
            if (n > 0) {
                received += static_cast<uint32_t>(n);
            } // Blocks until the entire feature vector is received
        }

        // ---- Run KNN classifier on the received features ----
        knn_cls_predict(ExtractedFeatures, output); // Executes the k-NN model using the received features

        // ---- Send prediction vector back to PC (int32) ----
        uint8_t *tx_ptr = reinterpret_cast<uint8_t*>(output);
        uint32_t sent = 0;
        while (sent < bytes_to_write) {
            ssize_t n = pc.write(tx_ptr + sent, bytes_to_write - sent); // Sends the raw prediction scores back to the host
            if (n > 0) {
                sent += static_cast<uint32_t>(n);
            } // Blocks until the full output array is sent
        }
    }
}
```

- Python Code:

```python
import os
import time
import struct
import numpy as np
import serial                         # Essential for communicating with the microcontroller over serial (UART)
import scipy.signal as sig

from mfcc_func import create_mfcc_features  # senin modülün
```

```
# ==== KLASÖR AYARLARI ====
PROJECT_ROOT = os.path.dirname(os.path.abspath(__file__))
FSDD_PATH = os.path.join(PROJECT_ROOT, "recordings")  # Directory where audio test files are stored

# ==== MFCC PARAMETRELERİ (STM32 tarafıyla aynı) ====
FFTSize = 1024
sample_rate = 8000
numOfMelFilters = 20
numOfDctOutputs = 13     # Defines the base number of MFCCs (must match the model training)
window = sig.get_window("hamming", FFTSize)

# ==== VERİYİ YÜKLE (önceki kodunla aynı mantık) ====
recordings_list = [os.path.join(FSDD_PATH, rec) for rec in os.listdir(FSDD_PATH)]
test_list  = {rec for rec in recordings_list if "yweweler" in os.path.basename(rec)} # Defines the specific test set
train_list = set(recordings_list) - test_list

_, _ = create_mfcc_features(train_list, FFTSize, sample_rate,
                            numOfMelFilters, numOfDctOutputs, window)
test_mfcc_features, test_labels = create_mfcc_features(test_list, FFTSize, # Generates the test features (MFCCs + deltas)
                                                       sample_rate,
                                                       numOfMelFilters,
                                                       numOfDctOutputs,
                                                       window)

test_mfcc_features = np.asarray(test_mfcc_features, dtype=np.float32) # Converts features to the float32 type expected by the
MCU
test_labels = np.asarray(test_labels, dtype=np.int32)

num_features = numOfDctOutputs * 2   # Total number of features (e.g., 26)

# ==== SERIAL BAĞLANTI ====
# COM portu kendine göre değiştir (Windows: "COM5", Linux: "/dev/ttyACM0")
ser = serial.Serial(
    port="COM6",                         # Specify the COM port connected to the embedded board
    baudrate=115200,                     # Baud rate must match the C code's setting
    bytesize=serial.EIGHTBITS,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    timeout=1,
)

time.sleep(2.0)  # Gives the board time to reset and initialize

NUM_CLASSES = 10  # Number of classes (e.g., digits 0-9)

correct = 0
total = 0

for feats, label in zip(test_mfcc_features, test_labels): # Loops through each test sample
    # features boyutu kontrol
    if feats.size != num_features:
        print("Feature length mismatch:", feats.size)
        continue

    # ---- FEATURES → BOARD ----
    ser.write(feats.astype("<f4").tobytes())   # Sends features as little-endian float32 bytes

    # ---- BOARD → PREDICTION ----
    bytes_expected = NUM_CLASSES * 4  # 4 bytes per int32
    rx = b""
    while len(rx) < bytes_expected:
        chunk = ser.read(bytes_expected - len(rx)) # Reads bytes until the full prediction vector is received
        if not chunk:
            break
        rx += chunk

    if len(rx) != bytes_expected:
        print("Timeout / incomplete read from board")
        continue

    # int32 prediction vector
    preds = struct.unpack("<" + "i" * NUM_CLASSES, rx) # Decodes the little-endian int32 bytes into the prediction array

    pred_class = int(np.argmax(preds)) # Finds the class index with the highest prediction score
    total += 1
    if pred_class == int(label):
        correct += 1 # Tallies correct prediction

    print(f"True: {label}, Pred: {pred_class}, Raw: {preds}")

ser.close()

if total > 0:
    acc = correct * 100 / total
    print(f"¥nAccuracy over UART/STM32: {correct} * {100} / {total} = %{acc:.3f}") # Calculates and prints the final performance
metric
else:
    print("No valid samples tested.")
```

Part-3:

The Python script (Host PC) manages the test data (MNIST images), serial communication, and evaluation. It sends the raw 784-byte image data over serial to the target board and receives the single-byte prediction result.

The C code (Embedded Target) is the low-latency inference engine: it continuously receives the image bytes, performs the computationally intensive Hu Moments feature extraction, runs the pre-trained Decision Tree (DT) classification on those 7 features, and sends the final predicted class (0-9) back to the host for accuracy calculation.

Note: For this part, we included **hdr_feature_extraction.c** (in addition to **main.cpp**) because we **modified it** for this section, rather than using the **unchanged supplied code** from the first two parts.

Our Codes:

- C Code (**main.cpp**) :

```
#include "mbed.h"
// Ensure you include arm_math.h if your library relies on it,
// otherwise standard math.h is fine for the corrected feature extraction
#include "lib_image.h"            // Library likely containing image data structure definitions
#include "hdr_feature_extraction.h"// Contains functions to compute image features (e.g., moments)
#include "dt_cls_inference.h"     // Contains the pre-trained Decision Tree classifier function

#define BAUD_RATE 115200
#define IMG_SIZE 784 // 28x28           // Defines the size of the input image buffer (28 * 28 pixels)

// Use the specific pins for your board (DISCO-F746NG uses USBTX/USBRX)
UnbufferedSerial pc(USBTX, USBRX);
uint8_t img_buffer[IMG_SIZE];      // Buffer to store the incoming 28x28 grayscale image bytes

// Objects
IMAGE_HandleTypeDef img_handle;    // Structure to manage image metadata (width, height, format)
HDR_FtrExtOutput features;

// Feature array for DT
float hu_moments_array[7];        // Array to hold the 7 Hu Moments (invariant shape features)
// DT returns votes/probabilities (array of size 10)
int class_votes[10];             // Array to store classification results for each of the 10 digits (0-9)

int main() {
    pc.baud(BAUD_RATE);

    // Init Image Struct
    img_handle.pData = img_buffer;
    img_handle.width = 28;
    img_handle.height = 28;
    img_handle.format = IMAGE_FORMAT_GRAYSCALE;
    img_handle.size = IMG_SIZE;

    while (true) { // Main loop for continuous inference
        // 1. Sync - Send 'R' to tell Python we are ready
        char ready = 'R';
        if (pc.writable()) {
            pc.write(&ready, 1);    // Sends synchronization signal to the Python host
        }
        // 2. Receive Image (FIXED LOGIC)
        char* ptr = (char*)img_buffer;
        int remaining = IMG_SIZE;

        while (remaining > 0) {
            if (pc.readable()) {
                // Use ssize_t to capture error codes
                ssize_t n = pc.read(ptr, remaining); // Reads incoming image data bytes over serial

                // Only update pointers if data was actually read (n > 0)
                if (n > 0) {
                    remaining -= n;
                    ptr += n;
                }
            }
        } // Blocks until all 784 image bytes are received

        // 3. Extract Features
        hdr_calculate_moments(&img_handle, &features);
        hdr_calculate_hu_moments(&features); // Computes the 7 Hu Moments (shape features) from the image

        // 4. Map features to array
        for(int i=0; i<7; i++) {
            hu_moments_array[i] = features.hu_moments[i]; // Copies features into the input array for the classifier
        }

        // 5. Inference (Decision Tree)
        // Zero out votes before prediction just in case
        memset(class_votes, 0, sizeof(class_votes));
        dt_cls_predict(hu_moments_array, class_votes); // Executes the Decision Tree classification

        // 6. Argmax (Find predicted class)
        int best_class = 0;
        int max_vote = -1;

        for(int i=0; i<10; i++) {
            if (class_votes[i] > max_vote) {
                max_vote = class_votes[i];
                best_class = i;
            }
        } // Determines the predicted digit (0-9) based on the highest vote count

        // 7. Send Result
        char res = (char)best_class;
        pc.write(&res, 1);           // Sends the single-byte prediction back to the Python host
    }
}
```

- C Code (**hdr_feature_extraction.c**):

```
/*
 * hdr_feature_extraction.c
 * - Fixed Row-Major indexing
 * - Implemented Double Precision    // Notes on crucial optimizations for embedded accuracy
 * - Implemented Log Transform for Hu Moments
 */

#include "hdr_feature_extraction.h"
#include <math.h>                    // Required for log10() and fabs()
#include <string.h>                  // Required for memset()

// Helper for integer power to avoid expensive powf() and maintain precision
static double fast_pow(double base, int exp) // Optimized power function for small integer exponents
{
    if (exp == 0) return 1.0;
    if (exp == 1) return base;
    if (exp == 2) return base * base;
    if (exp == 3) return base * base * base;
    return 1.0;
}

int8_t hdr_calculate_moments(IMAGE_HandleTypeDef * img, HDR_FtrExtOutput *output)
{
    if (img->format != IMAGE_FORMAT_GRAYSCALE)
    {
        return -1;
    }

    // Use double for intermediate calculations to prevent precision loss
    double temp_moments[4][4];       // Array to store Raw Moments M_pq (p+q <= 3)
    memset(temp_moments, 0, sizeof(temp_moments));

    // --- 1. Calculate Raw Moments (M_pq) ---
    // Iterate Row-Major (Y then X) to match Python/OpenCV memory layout
    for(uint32_t y = 0; y < img->height; y++)
    {
```

```c
        for (uint32_t x = 0; x < img->width; x++)
        {
            uint8_t pixel = img->pData[y * img->width + x]; // Correct Row-Major indexing (y * width + x)

            // Thresholding: Matches Python cv2.threshold(128, 255, THRESH_BINARY)
            // If > 128, treat as 1.0 (Normalized binary mass)
            double pixel_val = (pixel > 128) ? 1.0 : 0.0; // Converts image to binary mass representation

            if (pixel_val > 0.0)
            {
                for(int i = 0; i < 4; i++)
                {
                    for(int j = 0; j < 4 - i; j++)
                    {
                        // Accumulates the raw moments M_pq = sum(x^p * y^q * pixel)
                        temp_moments[i][j] += fast_pow((double)x, i) * fast_pow((double)y, j) * pixel_val;
                    }
                }
            }
        }
    }

    // --- 2. Calculate Centroid ---
    if (temp_moments[0][0] == 0.0) return -1; // M_00 is the total mass/area

    // Centroid (x_c, y_c) = (M_10/M_00, M_01/M_00)
    double centroid_x = temp_moments[1][0] / temp_moments[0][0];
    double centroid_y = temp_moments[0][1] / temp_moments[0][0];

    // --- 3. Calculate Central Moments (mu_pq) ---
    // Formulas correct the Raw Moments for translation (centered at the centroid)
    double mu[4][4];
    memset(mu, 0, sizeof(mu));

    // Formulas for mu_pq (p+q <= 3) are derived from moment theory.

    // --- 4. Calculate Scale Invariant Moments (nu_pq) ---
    double area = mu[0][0];
    double norm_factor_2 = area * area;
    double norm_factor_3 = norm_factor_2 * sqrt(area); // Normalization factor for higher-order moments (p+q=3)

    // Store in output struct (cast to float is safe here)
    output->nu[2][0] = (float)(mu[2][0] / norm_factor_2); // nu_pq = mu_pq / M_00^gamma
    // ... other nu_pq assignments

    return 0;
}

void hdr_calculate_hu_moments(HDR_FtrExtOutput *output)
{
    // Use doubles for calculation
    // ... nu_pq initialization ...

    double h[7]; // Array to hold the 7 Hu Moments

    // Standard Hu Moment Formulas (7 invariant shape features, computed from nu_pq)
    h[0] = n20 + n02;

    h[1] = fast_pow(n20 - n02, 2) + 4 * fast_pow(n11, 2);

    // ... formulas for h[2] through h[6] ...

    // --- LOG TRANSFORM ---
    // This is crucial for scaling features to a range the Decision Tree can handle reliably.
    // Formula: -1 * sign(h) * log10(abs(h))
    for (int i = 0; i < 7; i++)
    {
        if (h[i] != 0.0)
        {
            double val = h[i];
            double sign = (val > 0.0) ? 1.0 : -1.0;
            // Use log10 of absolute value, flip sign to keep order
            h[i] = -1.0 * sign * log10(fabs(val)); // Applies the log transformation
        }
        else
        {
            h[i] = 0.0;
        }

        // Assign final transformed feature to output struct
        output->hu_moments[i] = (float)h[i]; // Stores the final feature vector used by the Decision Tree
    }
}
```

- Python Code:

```
import serial                       # Essential for serial communication with the embedded device
import time
import numpy as np
import struct
import sys
import os
from mnist import load_images, load_labels

# --- CONFIG ---
SERIAL_PORT = 'COM6'
BAUD_RATE = 115200                  # Baud rate must match the C code on the microcontroller
# NOTE: Use the exact same IMG_SIZE as defined in your C code
IMG_SIZE = 784                      # 28 * 28 pixels

print("Loading Test Data...")
DATA_DIR = "MNIST-dataset"
test_images = load_images(os.path.join(DATA_DIR, "t10k-images.idx3-ubyte"))
test_labels = load_labels(os.path.join(DATA_DIR, "t10k-labels.idx1-ubyte"))

# Test 500 images
indices = np.random.choice(len(test_images), 500, replace=False) # Selects a random subset of 500 images to test

try:
    ser = serial.Serial(SERIAL_PORT, BAUD_RATE, timeout=2) # Opens the serial port
except:
    print("Error opening serial port")
    sys.exit()

ser.reset_input_buffer()
print("Waiting for MCU...")
while ser.read(1) != b'R':          # Waits for the 'R' synchronization signal from the MCU
    pass
print("Synced!")

correct = 0
total = 0

for i, idx in enumerate(indices):
    img = test_images[idx]
    label = test_labels[idx]

    # 1. Handshake
    if i > 0:
        while ser.read(1) != b'R':  # Waits for the MCU to finish the previous task and signal readiness ('R')
            pass

    # 2. Send Image (THE FIX)
    # The MCU C-code reads data in Column-Major order (c * height + r).
    # Python stores data in Row-Major order.
    # We must Transpose (.T) the image so the byte stream matches the MCU's reading logic.
    # This ensures the MCU calculates features on a "Normal 7", not a "Flipped 7".

    #img_for_mcu = img.T  # <--- THIS IS THE KEY FIX
    ser.write(img.tobytes())        # Sends the entire 784-byte image array to the MCU

    # 3. Receive Prediction
    resp = ser.read(1)              # Reads the single-byte prediction result from the MCU
    if not resp:
        break

    pred = int.from_bytes(resp, 'little') # Converts the received byte back into an integer prediction

    if pred == label:
        correct += 1
    total += 1

    if i % 50 == 0:
        print(f"Processed {i} | Acc: {correct/total*100:.2f}%") # Reports running accuracy periodically

print("-" * 30)
print(f"Final Accuracy: {correct/total*100:.2f}%") # Final performance metric for the entire embedded system
print("-" * 30)
ser.close()
```