Assignment Details

****

Any future updates will be highlighted in red

****

1. Background Concepts

In statistics, the **k-nearest neighbors** algorithm (**k-NN**) is one of the popular non-parametric classification methods. It is used for classification and regression. In this assignment, we are applying k-NN to solve a classification problem.

The input consists of the k closest training examples in a data set. In k-NN classification, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If k = 1, then the object is simply assigned to the class of that single nearest neighbor.

Illustrated by Figure-1, given two sets of data points which are labeled as **class A** (red) or **class B** (green) respectively, and **k** is chosen to be **3**. By calculating the euclidean distance (**d**) between each point and the **new example** (yellow), we can locate the 3 nearest neighbors (circled). There are **2** green samples and **1** red sample among the 3 nearest neighbors, the **new example** can be classified as **class B**, as class B (green) has a higher vote.

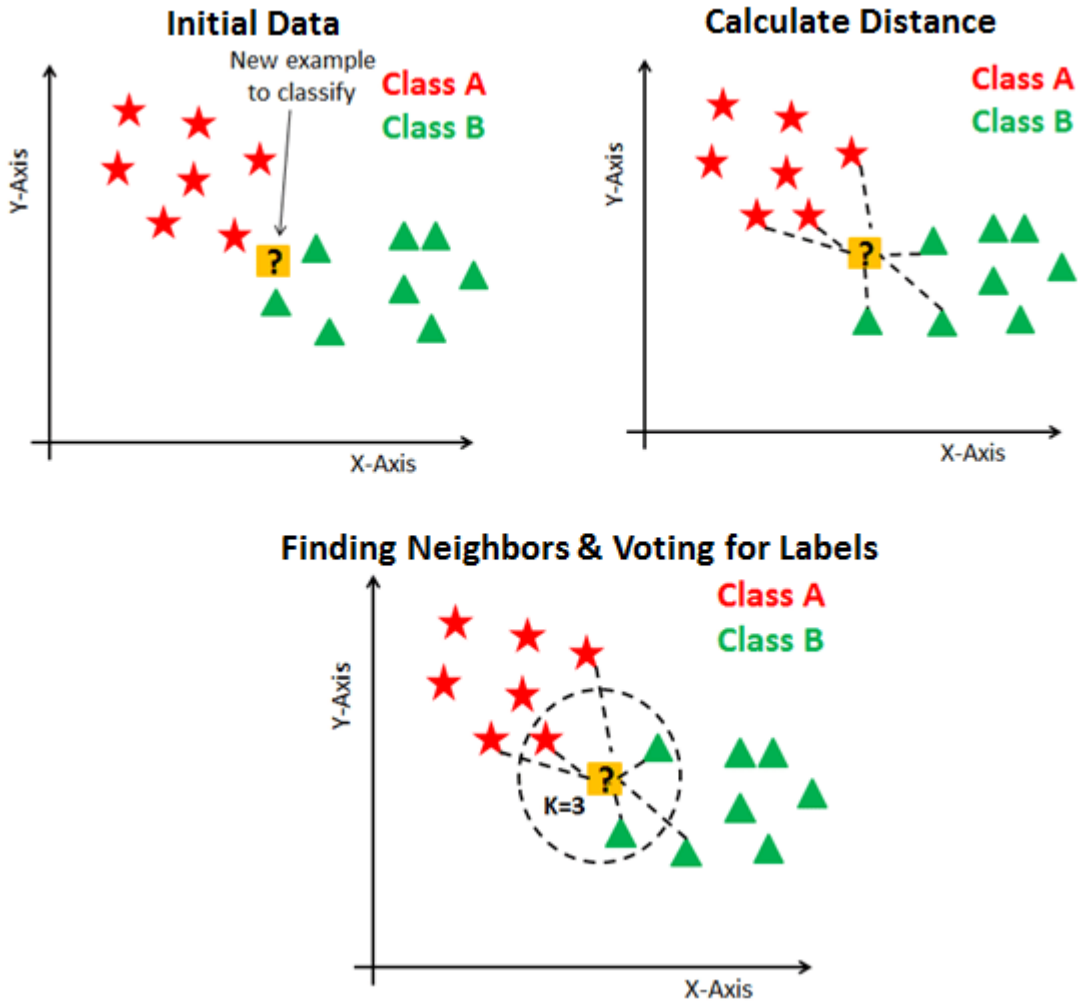

Figure-1. k-NN Algorithm for Binary Classification (binary classification refers to the classification of all data points into 2 groups).

(image courtesy: https://towardsdatascience.com/knn-visualization-in-just-13-lines-of-code-32820d72c6b6)

One of the most common distance metrics used in the k-NN algorithm is the **squared Euclidean distance** (d^2), The d^2 between two points can be easily calculated with the following equation:

$$d^2=(x_1 - x_2)^2 +(y_1 - y_2)^2$$

where:

- (x_1 , y_1) are the coordinates of sample point 1,
- (x_2 , y_2) are the coordinates of sample point 2,
- d^2 is the squared Euclidean distance between these two points.

2. Objectives

The objective of this assignment is to develop an ARMv7-M assembly language function that implements the function **int classification**(**int** N, **int*** points, **int*** label, **int*** sample) that classify the data points based on their k nearest neighbor votes.

where,

- **N** is the number of labeled training data in points.
- **points[][]** is an array that contains 10 times of x and y coordinates of the N number of training data.
- **label[]** is an array that contains the label of points in **points[][]**. They are either 1 or 0.
- **sample[]** contains 10 times of x and y coordinates of the test data, this is the sample point to be classified.
- the **classification()** function should return an integer, **class**, which is the predicted class of the sample data, it should be either 1 or 0.

You will also need to write down the machine code (8-digit hexadecimal, of the form 0xABCD1234) corresponding to each assembly language instruction as a comment next to the instruction in your assembly language function.

Note the following:

- You can assume that the instruction memory starts at a location 0x00000000.
- Follow the encoding format given in Lecture 4. The actual ARMv7M encoding format is different, but we will be sticking to the encoding format given in Lecture 4 for simplicity.
- Only the machine codes for the following instructions need to be provided.
    - Data processing instructions such as ADD, SUB, MOV[$], MUL, MLA, AND, ORR, CMP, etc., if they are used without shifts (i.e., the Operand2 is either a register without shift or imm8).
    - Load and Store instructions in offset, PC-relative, pre-indexed and post-indexed modes.
    - Branch instructions - conditional and unconditional, i.e., of the form B{cond} LABEL.

  [$]MOV is also one of the 16 DP instructions with the cmd 0b1101 as mentioned in slide 37 of Chapter 4. For MOV instruction, Rn is not used. You can encode Rn ($Instr_{19:16}$) = 0b0000. This makes sense as MOV has only one source operand which can be a register or immediate (recall: the assembly language format for MOV is MOV Rd, Rm or MOV Rd, #imm8), which means it can only come from the second source operand. Hence, the first source operand (which has to be a register, not immediate) is not used.

- You do not need to provide machine codes for instructions that do not fall into the categories above, even if you have used them in your program (e.g., BX, MOVW, multiplication instructions with 64-bit products such as UMULL/SMULL/UMLAL/SMLAL/SDIV/UDIV, division, data processing instructions where Operand2 is a register with shift etc.).
- Assume all instructions are 32 bits long, and that the assembler places the instructions and data (constants declared using .word) in successive word locations in the same order as they appear in assembly language. This should help compute the offset for PC-relative instructions, branches etc. Note that PC-relative mode is nothing but offset mode (PW=0b10) with Rn=R15.

You will also need to provide **a paper design showing how the microarchitecture** (datapath and control unit) covered in Lecture 4 can be modified to support MLA and MUL instructions. You can assume that a hardware multiplier block is available, which takes in two 32-bit inputs Mult_In_A and Mult_In_B, and provides a 32-bit output, Mult_Out_Product combinationally (i.e., without waiting for a clock edge).

3. Procedure

(a) Preparations

After completing the Self Familiarisation, you should have the “Lib_CMSISv1p30_LPC17xx” project and several other projects, in the Project Explorer pane of the LPCXpresso IDE (LXIDE).

The “Lib_CMSISv1p30_LPC17xx” project contains the Cortex Microcontroller Software Interface Standard (CMSIS) files.

Download Assignment 1 template which contains the “CG2028AssignS1AY2122.zip” (updated 26/10/21) project and unzip it into your workspace.

• **classification.s**, which presently contains only a few instructions – this is where you will write the assembly language instructions that implement the **clas sification()** function to classify the data points into 2 groups; and

• **main.c**, which is a C program (to be described below) that calls the **classification()** function with the appropriate parameters and prints out the result on the console pane. You do NOT need to modify this file unless you want to test your assembly language function with different values in the arrays or a different number of data points N.

(b) Initial Configuration of Programs

The C program **main.c** defines a 2D array ***points[N][2]*** that contains the coordinates of **N** number of data points, and a 1D array ***label[N][2]*** that defines the class of the corresponding data points. In order to do computation using functions written by assembly instructions, the coordinates are multiplied 10 times larger to pass into the functions. Therefore, think of the value 100 in ***points[N][2]*** as 10.0. A 1D array ***sample[3]*** defines the coordinates of the test data and the number of neighbors, ***k***.

Figure-2 provided an illustration of ***points[N][2]***, ***label[N][2]*** and ***sample[3]***.

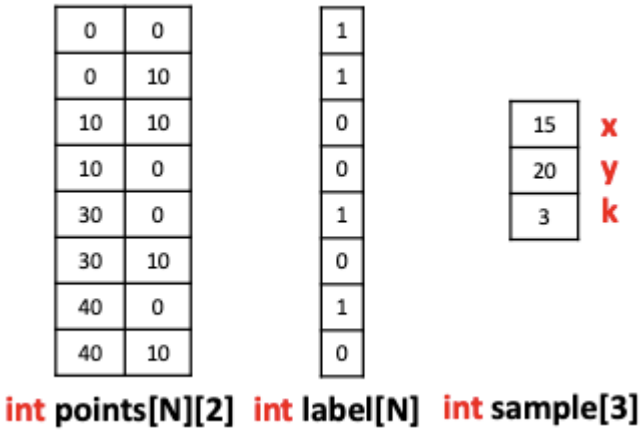

Figure-2. Illustration of elements of the array points[][] and class[][].

**Note: The actual values for each array in the given case may be different from the figure shown here.*

The elements of a 2D array are stored row by row in **consecutive words** in memory starting from word address that can be found by its pointer value, illustrated in Figure-2. For example, the starting address of ***points[][]*** can be found by its pointer **(int*)points10**.

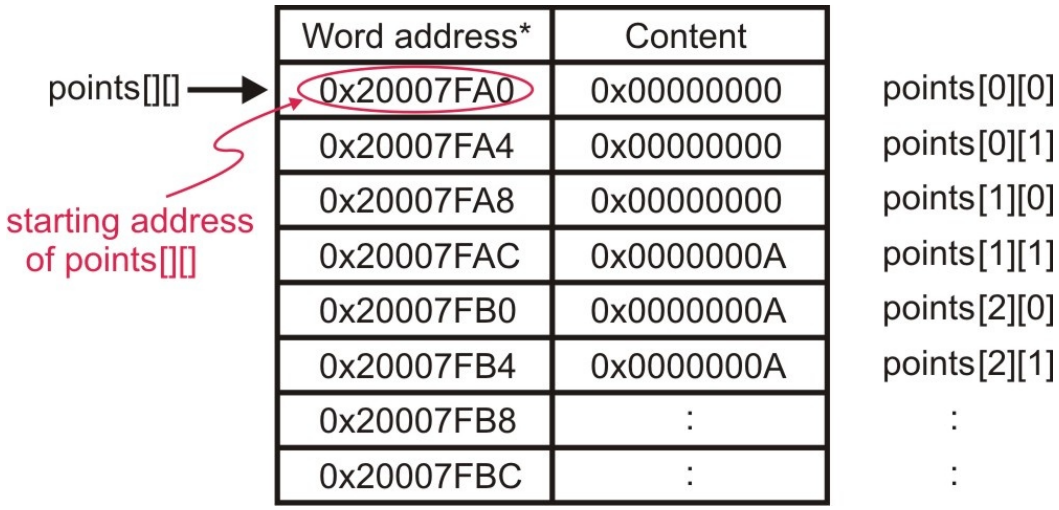

Figure-3. Illustration of contents of consecutive memory locations containing elements of the array points[][].

**Note: The actual memory addresses for each array in your case may be different from the figure shown here.*

The C program calls the function **classification()** written by assembly language to determine the class of the sample point and returned as an integer.

Note: The functions **classification()** that you develop should be able to perform the required computation for any reasonable value of **N** (number of training data points) and k (number of neighbors to be considered).

For this assignment, you may assume the distances between the sample data and each training point are unique so that you could always find the nearest k points, and k is always 1.

Parameter passing between C program and assembly language function

In general, parameters can be passed between a C program and an assembly language function through the registers. In the function **extern int asm_func (arg1, arg2, ….)**:

arg1 will be passed to the assembly language function asm_func() in the **R0** register, **arg2** will be passed in the **R1** register, and so on. In total, 4 parameters can be passed from the C program to the assembly program in this way.

If there is a **return value** from the assembly language function back to the C program, the **R0** register should be used to keep the return value right before the program comes back to **main.c**.

In the function written by assembly language, there is a subroutine called **SUBROUTINE** declared after the main part of the assembly program, in order to demonstrate the way to "create and call a function" in an assembly language program. *Note*: The purpose of this declaration is to lead to the thinking of link register (LR/R14). At the completion of this assignment, **SUBROUTINE** is not compulsory to be used, i.e. you may not have BL SUBROUTINE in your function **asm_fun()**.

(c) Procedure

Compile the "CG2028AssignS2AY202122" project and execute the program.

Explain the console window output that you see.

Write the code for the assembly language function **classification**().

The assembly language program only needs to deal with integers.

Verify the correctness of the results computed by the function you have written that appears at the console window of the LXIDE. A C language function **classification_c**() is provided as a reference.

Please note that the C function is provided just as a reference, and is NOT optimized. You are encouraged to implement a more optimized assembly language program.

There are several aspects that you need to pay attention to:

- It is a good practice to push the contents in the affected general-purpose registers onto the stack upon entry into the assembly language function, and to pop those values at the end of the assembly language function, just before returning to the calling main program.
- In a RISC processor such as the ARM, arithmetic and logical operations only operate on values in registers. However, there are only a limited number of general-purpose registers, and programs, e.g. complex mathematical functions, may have many variables.

Hint: Use and reuse the registers in a systematic way. Maintain a data dictionary or table to help you keep track of the storage of different variables in different registers at different times.

Show the assembly language program and actual console output to the Graduate Assistant (GA) during the assessment session.

4. Assessment

For each group of 2 students:

- Write a short report of about 4 pages long to explain how your assembly language program works, as well as the machine codes and microarchitecture design described in Section II above.
- For more information on the assessment schedule and process see the "Assessment and Submission info" tab

Some details about assessment:

- The assessing GA will modify your program slightly to see if your assembly language code is implemented correctly.
- The GA will verify the machine codes for each instruction (that you wrote as comments) during the assessment.
- You should also show and explain the paper design of your microarchitecture incorporating MUL and MLA instructions to the assessing GA. The design can be hand-drawn as long as it is legible – there is no need to spend your time on computer drawn schematics. Only the parts you have modified needs to be shown (redrawing the schematic already given in the lecture notes is not necessary).
- The Assignment mark for each student will consist of a group component and an individual component. The individual component will be based on a peer review as well as the perception of the assessing GA.
- The assessing GA(s) will ask each student some questions in turn during the assessment. Both students need to be familiar with all aspects of the assignment and your solution as the GA can choose any one of you to explain any part of the assignment. The GA will also ask you additional questions to test your understanding.
- To test your understanding of instruction encodings / machine codes, the GA could also ask you to figure out the encoding of any instruction whose format is specified in Lecture 4 even if you have not used that particular instruction in your program.

END