# Project 2: Boston Housing Model Evaluation Program

(400 points)

You are a mainframe developer working for a nation-wide corporation contracted by the City of Boston's Civil Planning and Development Office to develop and manage the parts of the city's IT infrastructure dedicated to urban development planning. Your division has been tasked with developing a business intelligence module for the system, and you have been given the task of writing the code for a submodule that evaluates a set of models generated by the core components of the intelligence software as new data becomes available. The models your program will test are the parameters of a single-node neural network (or neuron) created by the main software that performs a linear regression, and must be tested on data containing known combinations of inputs and outputs to get an idea of how well they could be expected to generate accurate predictions on unseen data.

The program you will write will take a set of models, each consisting of the parameters of the prediction neuron (weights on inputs and a bias), generate a set of predictions using the housing data with each and then calculate the Mean Squared Error (MSE) and $\mathrm{R}^2$ statistics for those models on three randomly-sampled subsets of the Boston Housing data (which will be provided in separate tables in the input stream).

## Primer: The neuron

Deep neural networks (or DNN's) provide the backbone architecture for some of the most advanced and capable AI technologies in use today, dominating such cutting-edge application domains as computer vision and Natural Language Processing (NLP). A large part of its appeal (and its power) lies in its simplicity. Deep neural networks are based on a very old concept developed in the 1950's called the artificial neuron.

Figure 1 shows a single neuron having index $j$, with inputs $x_1 \ldots x_n$ and a bias $b_j$ specific to the neuron. The inputs are aggregated using the formula shown ($\sum \mathbf{wx} + b_j$), which is the linear sum of the inputs multiplied by a corresponding set of learned weights $w_1 \ldots w_n$ and offset by

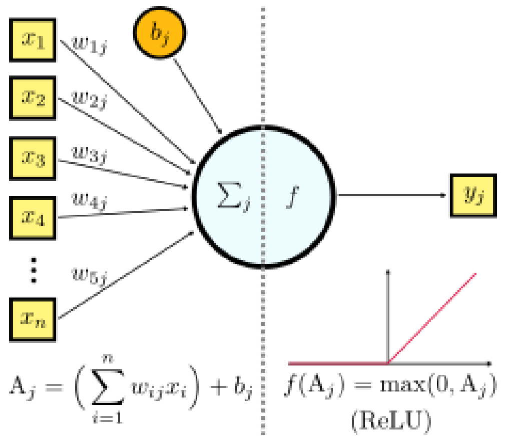
$x_1$
$x_2$
$x_3$
$x_4$
$x_n$
$w_{1j}$
$w_{2j}$
$w_{3j}$
$w_{4j}$
$w_{5j}$
$b_j$
$\sum_j$
$f$
$y_j$
$\mathrm{A}_j = \left(\sum_{i=1}^{n} w_{ij}x_i\right) + b_j$
$f(\mathrm{A}_j) = \max(0, \mathrm{A}_j)$
(ReLU)

Figure 1: A single neuron $j$. Inputs $x_1 \ldots x_n$ are aggregated and the result passed through an activation function $f$, yielding output $y_j$.

the neuron's bias $b_j$, which is also learned during training [1]. The result $\mathrm{A}_j$ is passed through an activation function $f$ to generate the final output $y_j$. There are several different activation functions that we could use, each with its own strengths and weaknesses. By far the most popular and commonly used is the Rectified Linear Unit (ReLU) function, also shown in figure 1. It's also by far the simplest to implement so we'll use it here.

In the context of deep learning, neurons are simply called nodes, and a cursory glance at figure **??** should make the reasoning fairly self-evident. Figure **??** shows a simple feed-forward network, so called because processing propagates from a set of inputs through a set of layers containing multiple nodes in order, transforming the inputs to a set of outputs [2]. In the first layer, all nodes receive the same inputs $x_1 \ldots x_n$. Each node has its own set of weights and a bias that are used during aggregation. Nodes are "activated" when the activation function yields a non-zero result, and the outputs generated by all the nodes in one layer (such as the layer receiving the inputs) become the inputs to the next.

Neural networks let us perform extremely useful tasks on complex data, which would otherwise be extremely difficult and computationally intensive if not impossible [3]. In this project, we will use neural networks to implement a nonlinear regressor that performs predictions on the Boston Housing data set. We'll be using pre-trained parameters, so the only complexity we'll have to deal with will be implementing the feed-foward algorithm, which is introduced below. Our regressor will generate predictions of the median values of homes based upon a number of factors such as adjacent zoning, crime statistics and distance from schools and shopping districts. We will compare the predictions our regressor makes using different sets of parameters (models) with the actual values collected when the Boston housing survey was originally performed to see how well the regressor models perform.

## Program Requirements

### Table-Building (125points)

#### Data Set Tables

Your program will first need to load three sets of data, all of which are random samples from the Boston Housing dataset, which it will later use to evaluate the performance of the regressor models. Each row in a data set describes a sample, or instance of some object or system of interest, while each column quantifies some property (called a feature) of the object or system. The Boston Housing data set has 13 features (columns) which are used to predict a median value for a home or property. The Boston Housing data set also includes target property median values, known in machine learning as *ground truth values* (denoted by the vector $\hat{\mathbf{y}}$), which the regressor models attempt to duplicate as closely as possible. For those familiar with statistics, the feature columns are the independent variables while the ground truth column contains known values for a dependent variable [4]. Therefore each row of the Boston data set has 14 columns, with the first 13 containing data describing the attributes of a real-estate property (house) and

[1] We won't concern ourselves with the details of the algorithms used to train deep neural networks as they are *ridiculously* beyond the scope of anything we'll need for this project.

[2] For mathematical reasons, the sets of inputs and outputs are usually referred to as the input and output tensors, which to us coders is basically just a fancy term for a resizable array.

[3] It has even been shown possible to solve the infamous traveling salesman problem in constant time using quantum neural networks regardless of the size of the input graph.

[4] The difference between predicted and ground truth values $(\mathbf{y} - \hat{\mathbf{y}})$ are called the *residuals*, which we've seen before and will become important during model performance evaluation.

the 14th as a target median value. Table 1 shows the formatting of the data set.

Table 1: The Boston Housing data set

| Description | Feature columns ($x_1, x_2, ..., x_{13}$) | Ground truth/Target ($\hat{y}$) |
|---|---|---|
| Storage footprint | 13 × PL8 | PL8 |

In our case, this means that later when your program processes the samples in the tables using the feed-forward algorithm, the feature columns will be used as the inputs $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$ and the results will be compared to the values in the ground truth column to determine how closely a model is able to reproduce the expected answer. You may want to make your table rows large enough to hold 15 8-byte packed decimals; doing so will allow you to easily store predicted values alongside the ground truth values, both of which will be used later to calculate the residuals.

Write an external subprogram labeled BUILDDAT to build the data set tables. Your main program should pass it 1) the address of the start of the table to fill; and 2) the trailing address of the table so that BUILDDAT doesn't try to write past the end of the storage allocated for the table. Each table (in the input) will terminate with the usual '*' delimiter in the following record; if the trailing address of the table is encountered before the delimiter BUILDDAT should set the return code (in register 15) to 1 and exit. Testing for a non-zero return code and printing a suitable error message (hint: you can use character string literals with XPRNT) will allow you to easily debug your table declarations. Your main program should call BUILDDAT three times; once for each data set table in the input. Use of a DSECT with the table rows is highly recommended.

### The Model Table

Your program will also need to evaluate at least one model. Models are simply a set of weights (or coefficients) and a bias in a row of data in a record of input. As the neuron associates a weight with each input, the models will have exactly the same number of weights as there are columns in the input data table. Write an external subprogram labeled BUILDMDL that implements the code to build the models table. Note that because we're restricted to 80 characters/bytes per record of input, you can expect that your program will once again need to handle rows wrapping across several records.

| Description | Weight columns ($w_1 \ldots w_{13}$) | Bias column |
|---|---|---|
| Storage footprint | 13 × PL8 | PL8 |

## Neuron Processing Implementation (125 points)

Once the tables have been built, your program will need to process the inputs from all 3 data tables. Processing on the neuron is a very simple two-step process; we first perform a linear aggregation of the inputs by crossing them with the weights, and then pass the result through an activation function. Because we're using the neuron to perform simple linear regression, you'll use the linear activation function (which is the identity function $f(x) = x$). For those of you familiar with linear algebra, to perform aggregation on the inputs we simply compute the cross-product of the input and a weight vectors for a model $j$, yielding a vector with a single element of which we (notionally) take the L2 norm and add the neuron's bias as in Equation 1.

**Algorithm 1** The cross-product computation algorithm

```
Input: list of inputs $\mathbf{x}$, list of weights $\mathbf{w}$
Output: scalar $y$
procedure L2-CROSS-PRODUCT($\mathbf{x}$, $\mathbf{w}$)
  $y \leftarrow 0$
  for $x, w$ in $\mathbf{x}$, $\mathbf{w}$ do
    $y \leftarrow y + x \cdot w$
  end for
end procedure
```

Alternatively, we can describe the aggregation in terms of Equation 2, where we multiply each input by neuron $j$'s corresponding weight (e.g., $x_1 \cdot w_1 j$, $x_2 \cdot w_2 j$, and etc.) and sum the results together before adding neuron $j$'s bias to the result.

$$\|\mathbf{w}_j^{\mathrm{T}} \times \mathbf{x}\| + b_j \tag{1}$$
$$\left(\sum_i w_{ij} x_i\right) + b_j \tag{2}$$

Write an external subprogram called PROCTABL that takes the start address of a data table and the address of a row in the models table created by the BUILDMDL external subprogram. Your program should loop through the rows of the data table and for each sample, it should compute a prediction $\hat{y}$ using the weights and bias in the models table row. Algorithm 1 details a compact psuedocode implementation that you can follow when writing the code to perform the linear aggregation on the inputs. Define an internal subroutine called CRSSPROD inside PROCTABL and write your implementation there.

Since we're "using" the linear activation function on the neuron we can skip the processing on the $\hat{y}$ result since $f(x) = x$ is the identity function. You should have declared your data tables with enough space to store an extra 8-byte packed decimal (1 extra column) per row, so place each sample's $\hat{y}$ there.

## Evaluating Model Performance (75 points)

After your main program computes the predicted values for all three data sets it will need to evaluate model performance on each and report them. You will use two common regression analysis statistics to accomplish this: Mean Squared Error (MSE) and R$^2$. Both require your program to compute the residuals $(\mathbf{y} - \hat{\mathbf{y}})$ as a first step. The equations are given below:

$$\mathrm{MSE} = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2 \tag{3}$$

$$\begin{aligned} \mathrm{R}^2 = 1 - \frac{\mathrm{RSS}}{\mathrm{TSS}}, \text{ where } \mathrm{RSS} &= \sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2 \\ \mathrm{TSS} &= \sum_{i=1}^{n} \left(y_i - \bar{y}_i\right)^2 \end{aligned} \tag{4}$$

We have seen both the residual sum of squares (RSS) and the total sum of squares (TSS) in previous assignments, though perhaps not as computations on packed decimals so you may need to adjust your old code.

Listing 1: Model evaluation report format.

```
Dataset   MSE        R^2
      1   ...        ...
      2   ...        ...
      3   ...        ...
```

Write an external subprogram EVALMODL, and implement MSE and $R^2$ as internal subroutines (to EVALMODL). The EVALMODL subprogram should take the address of a single data set table containing the **y** and computed **y** columns and return the MSE and $R^2$ statistics as 10-byte packed decimals. Your main program should call EVALMODL for each data set table and store the MSE and $R^2$ results separately. It is recommended that you store them as tuples in a sequence; you'll need to pass them all to the next subprogram (called from the main program) that will display the results.

Finally, write an external subprogram DISPLAY that takes the MSE and $R^2$ statistics just computed in EVALMODL. Use the format in Listing 1 to display the results for each dataset.

## Extending to Multiple Models (75 points)

To earn all possible points for the project, you will need to process all of the models in the models table. You will need to write, or perhaps modify, code to loop over the rows of the model table, use PROCTABL on each model (and all three tables per model) and store the MSE's and $R^2$'s calculated for each. Update your results reporting to use the format in Listing 2 given below.

Listing 2: Model evaluation report format v2.

```
Model:
   Architecture:
     weight 1: ...
     weight 2: ...
     ...

   Evaluation:
     Dataset   MSE        R^2
           1   ...        ...
           2   ...        ...
           3   ...        ...
```