```
package com.demo.AI_Methods_CW;

//Importing all required libraries
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.*;

import org.apache.poi.xssf.usermodel.XSSFWorkbook;
import org.apache.poi.ss.usermodel.*;

public class Access_Spreadsheet {
	//Class imports and exports to an excel spreadsheet using Apache POI

	//Defines a string to hold the name of the file
	private static String NAME = new String();

	//Main method
	public static void main(String[] args) {
		Scanner scanner = new Scanner(System.in);
		//Takes a user input for the number of nodes for the ANN
		System.out.println("Enter number of hidden nodes: ");
		int numberofHiddenNodes = scanner.nextInt();

		//Calls method that works with excel file
		readinData(numberofHiddenNodes);

		scanner.close();
	}

	//Accesses excel spreadsheet and pulls data from the file to pass through MLP
class
	//Exports the returned data back to spreadsheet
	public static void readinData(int numHidden) {
		//Opens try/catch statement
		try {

			//Defines name of the file being accessed
			//This file contains all cleansed, randomised, and standardised
data
			NAME = "AI Methods Coursework/Data Set.xlsx";

			//Imports the file
			FileInputStream input_file = new FileInputStream(new File(NAME));

			//Defines the whole import as a workbook
			//Then defines a single sheet for the data set
			Workbook workbook = new XSSFWorkbook(input_file);
			Sheet sheet = workbook.getSheetAt(0);

			//Defines number of rows and number of columns
			int numberofRows = sheet.getLastRowNum() + 1;
			int numberofCols = sheet.getRow(0).getLastCellNum();

			//Define a 2D float array for each row in the data set to be
added to
			float[][] dataSet = new float[numberofRows][numberofCols];

			//Restricts data imported to the number of training data rows
```

```
			//Value is changed to 263 for both validation and testing data
			for (int i=0; i<789; i++) {
				//Iterates through every row of data
				Row row = sheet.getRow(i);

				//Restricts data to the predictors and predictand column
				for (int j=0; j<6; j++) {
					//Iterates through each column on a row

					//Defines a cell object
					Cell cell = row.getCell(j);
					//Converts the cell data from a cell object to string
					String cellValue = cell.toString();
					//Converts the string to a float and adds to array
					dataSet[i][j] = Float.parseFloat(cellValue);

				}

			}

			//Array for modelled data
			//Calls on main method from MLP class
			float[] modelledData = MLP.main(dataSet, numHidden);

			//Sets cells in a new column for the new prediction values
			for (int j=0; j<numberofRows; j++) {
				Row row = sheet.getRow(j);
				row.createCell(numberofCols).setCellValue(modelledData[j]);
			}

			input_file.close();

			//Uses an output stream to write back the new data to the 
spreadsheet
			FileOutputStream output_file =new FileOutputStream(new 
File(NAME));

		workbook.write(output_file);
		output_file.close();

		workbook.close();

		}
		catch(Exception e) {
			e.printStackTrace();
		}
	}



}
```

```
package com.demo.AI_Methods_CW;

//Importing all required libraries
import java.util.Random;

public class MLP {

	//Method to return a random value between an interval
	//Used for biases and weights
	public static float returnRandomIntervalValue(int numInputs) {
		//Sets minimum and maximum values for interval
		float max = 2/numInputs;
		float min = -2/numInputs;

		//Returns a value between 0 and 1
		Random r = new Random();
		//Alters number to required interval
		float randomNumber = min + r.nextFloat() * (max - min);
		return randomNumber;


	}

	//Method that sets weights and biases at random
	public static float[][] setWeightsandBiases(int numInputs, int numHidden) {
		int Wi = numInputs + numHidden + 2; //Size for input nodes, hidden
nodes, one bias array and one output node
		int Wj = Wi-1; //Maximum number of weights between nodes
		float[][] weights = new float[Wi][Wj]; //Weights and biases array

		//Sets random values for biases
		//Initialises to zero for positions corresponding to inputs
		for (int i=0; i<Wj; i++) {
			if (i<numInputs) {
				weights[0][i] = 0;
			} else {
				weights[0][i] = returnRandomIntervalValue(numInputs);
			}
		}

		//Sets random weights for input nodes to hidden nodes
		for (int i=1; i<numInputs; i++) {
			for (int j=0; j<Wj-1; j++) {
				if (j<numInputs) {
					//Input nodes are not connected to each other so are
initialised to zero
					weights[i][j] = 0;
				} else {
					weights[i][j] = returnRandomIntervalValue(numInputs);
				}
			}
			//The final position represents output node so will always be
zero
			weights[i][Wj-1] = 0;
		}

		//Sets random values for hidden nodes to output node
		for (int i=numInputs; i<Wj; i++) {
			for (int j=0; j<Wj-1; j++) {
```

```
                        //Hidden nodes only connect to the output node so other 
weights =0
                        weights[i][j] = 0;
                  }
                  //Weight from hidden node i to output node
                  weights[i][Wj-1] = returnRandomIntervalValue(numInputs);
            }

            return weights;
      }

      //Method to initialise activations
      //Called for every row so a separate method to setWeightsandBiases
      public static float[] setInitialActivations(int numInputs, float[] inputs, 
float[][] weights) {
            int Wj = weights[0].length;

            float[] activations = new float[Wj]; //Activations array
            for (int i=0; i<Wj; i++) {
                  if (i<numInputs) {
                        //Sets activations in input positions to input value
                        activations[i] = inputs[i];
                  } else {
                        //Initialises other positions equal to the bias for the 
given node
                        activations[i] = weights[0][i];
                  }
            }

            return activations;
      }

      //Main method
      public static float[] main(float[][] inputs, int numHidden) {
            int numberofRows = inputs.length; //Defines number of rows to iterate 
through
            int numberofInputs = inputs[0].length; //Defines number of inputs for 
given data set

            float[] modelledData = new float[numberofRows]; //Returned array of 
modelled data

            float[][] weights = setWeightsandBiases(numberofInputs, numHidden);

            //Defined variables for Bold Driver
            float sumError = 0;
            float currentMSE = 0;
            //Rho value for updating all vertices
            float p = 0.1f;

            //Main loop
            for (int epoch=0; epoch<=1; epoch++) {
                  //Iterates through all rows and passes through algorithm
                  for (int i=0; i<numberofRows; i++) {

                        //**BACK PROPOGATION ALGORITHM**
                        float[] activations = setInitialActivations(numberofInputs, 
inputs[i], weights);
```

```
                        activations = forwardPass(weights, activations,
numberofInputs);
                        float[] deltaValues = backwardPass(activations, weights,
numberofInputs, inputs[i][numberofInputs]);
                        weights = updateVertices(weights, deltaValues, activations,
p);

                        float uj = activations[activations.length - 1];
                        modelledData[i] = uj;

                        //Calculates error for bold driver modification
                        sumError = (float) (sumError + Math.pow((inputs[i]
[numberofInputs] - modelledData[i]),2));

                  }

                  //Every 2000 epochs, learning rate is updated
                  if (epoch % 2000 == 0 && epoch != 0){
                        float[] tempArray = calculateMSE(sumError, currentMSE,
numberofRows, epoch, p);
                        currentMSE = tempArray[0];
                        p = tempArray[1];
                  }
            }
            return modelledData;
      }

      //**Forward pass of BP Algorithm**
      public static float[] forwardPass (float[][] weights, float[] activations,
int numInputs) {
            int lenA = activations.length; //Sets the number of activations/nodes
            float[] weightedSums = new float[lenA];

            int lenW = weights.length; //Sets the number of total weights including
biases

            for (int i=numInputs; i<lenA; i++) {
                  for (int j=1; j<lenW; j++) {
                        //Iterates through all weights/biases and calculates each
weighted sum for every node, not inclduing input nodes
                        weightedSums[i] = weightedSums[i] + (weights[j][i] *
activations[j-1]);
                  }

                  //Adds biases on separately as only needs to be added once
                  weightedSums[i] = weightedSums[i] + weights[0][i];

                  //Uses sigmoid function to calculate new activation for each node
                  activations[i] = (float) (1/(1+ Math.exp(-weightedSums[i])));

            }
            return activations;
      }

      //**Backward pass of BP Algorithm**
      public static float[] backwardPass(float[] activations, float[][] weights,
int numInputs, float output) {
            int lenA = activations.length; //Sets the number of activations/nodes
            float[] deltaValues = new float[lenA];
```

```
		//Finds derivative and delta value for output node
		//Has a different formula
		float derivative = activations[lenA-1] * (1-activations[lenA-1]);
		deltaValues[lenA-1] = (output-activations[lenA-1]) * derivative;

		//Iterates backwards to find all other delta values for hidden nodes
		for (int j=lenA-2; j>numInputs-1; j--) {
			derivative = activations[j] * (1-activations[j]);

			for (int i=0; i<lenA; i++) {
				deltaValues[j] = (weights[j+1][i] * deltaValues[lenA-1]) *
derivative;
			}
		}

		return deltaValues;

	}

	//**Updating biases and weights**
	public static float[][] updateVertices(float[][] weights, float[] delta,
float[] activations, float learningRate){
		int lenW = weights[0].length;
		float p = learningRate; //Sets rho equal to learning rate passed
through method
		float a = 0.9f; //Defines alpha value for momentum modification

		for (int i=0; i<lenW; i++) {
			//Iterates through biases
			float newWeight = weights[0][i] + (p * delta[i] * 1);
			float momentum = a * (newWeight-weights[0][i]);

			weights[0][i] = newWeight + momentum; //Updates each bias
including momentum

		}

		for (int j=1; j<lenW+1; j++) {
			for (int k=0; k<lenW; k++) {
				if (weights[j][k] != 0) {
					//Updates existing weights using momentum
					float newWeight = weights[j][k] + (p * delta[k] *
activations[k]);
					float momentum = a * (newWeight-weights[j][k]);

					weights[j][k] = newWeight + momentum;
				}
			}
		}

		return weights;


	}

	//Bold Driver method
	//Finds the difference between two MSE values and updates learning rate
appropriately
```

```
	public static float[] calculateMSE(float sumError, float currentMSE, int numRows, int numEpochs, float learningRate) {
		float p = learningRate;

		float newMSE = sumError/numRows; //Calculates new mean squared error for current row

		//Adjusts learning rate
		if (newMSE > currentMSE * 1.10 && p > 0.01) {
			//If MSE value has increased by 10% and rho is within the interval
			//Rho decreases
			p = p * 0.7f;
		} else if (newMSE < currentMSE * 0.90 && p < 0.5) {
			//If MSE value has decreased by 10% and rho is within the interval
			//Rho increases
			p = p * 1.05f;
		}

		//Stored in a temporary array so both float values can be returned
		float[] temp = {newMSE, p};

		return temp;
	}

}
```