Table of Contents

ECG703 HW 2: Perceptron - James Skelly 1
Part A. Memory Allocation for Data Matrix 1
Part B. Populating the Data Matrix 1
Part C. Plotting the Target Function 2
Part D. Linear Separation and Plotting of Data 3
Part E. Implementing Perceptron Algorithm 6
Questions Regarding Homework Assignment 9
Functions 12
Final Comments & Conclusions 13

ECG703 HW 2: Perceptron - James Skelly

This program implements the perceptron algorithm for a set of randomly generated 2D data. The data is made "linearly separable" using a given target function. Prior to each section, a description will be given which contains both the instructions for the section and a description of the motivation and thought process behind the implementation.

```
format compact
clear all
close all
```

Part A. Memory Allocation for Data Matrix

Instructions: Create a set of 2D data (d = 2, features: X = (x1, x2)).

Implementation: A 20x2 matrix of zeros is generated for part A in order to allocate the necessary memory for the random data which will populate the matrix in part B.

```
X = zeros(20,2); % Allocate memory for the data matrix
x1 = X(:,1); % Assign all values in 1st column of X to x1 vector
x2 = X(:,2); % Assign all values in 2nd column of X to x2 vector
```

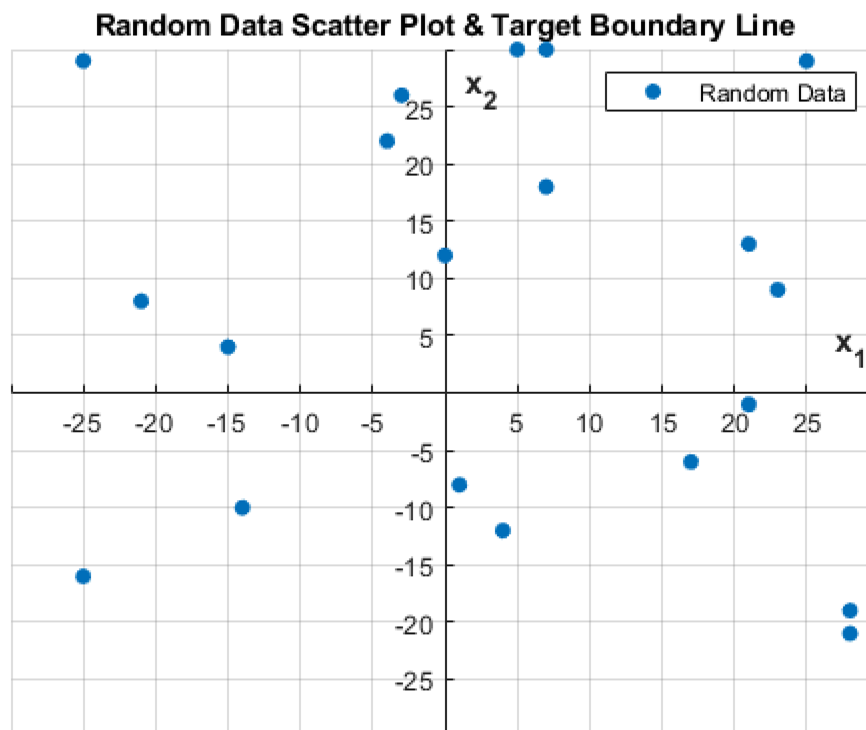
Part B. Populating the Data Matrix

Instructions: Randomly create a set of 20 data points (N = 20) such that for each point X = (x1,x2), the coordinates x1, x2 be integers; x1, x2 are to be limited to the [-30,+30] range and uncorrelated.

Implementation: A 20x2 matrix of random integers in the range specified is generated. The first column values are assigned to an x1 vector and the second column values are assigned to an x2 vector, where one point in the coordinate plane is of the form (x1,x2). The data is plotted as a scatter plot along with the target function from part C.

```
X = randi([-30,30],[20 2]); % Generates a random 20x2 matrix of
 integers
x1 = X(:,1); % Reassign all values in the first column of X to x1
 vector
x2 = X(:,2); % Reassign all values in the second column of X to x2
 vector
```

```
% Plot and axis settings, limits, labels for plot 1
figure('Name', 'HW2: Perceptron Plot 1')
scatter(x1,x2,'filled', 'DisplayName', 'Random Data');
legend
title('Random Data Scatter Plot & Target Boundary Line');
xlabel('x_{1}', 'FontSize', 12, 'FontWeight', 'bold');
ylabel('x_{2}', 'FontSize', 12, 'FontWeight', 'bold');
ax = gca;
ax.XAxisLocation = 'origin';
ax.YAxisLocation = 'origin';
xlim([-30 30]);
xticks(-30:5:30);
ylim([-30 30]);
yticks(-30:5:30);
grid on
hold on % Plot the line and scatter plot on the same figure
        % Scatter plot is plotted in part C, next
```



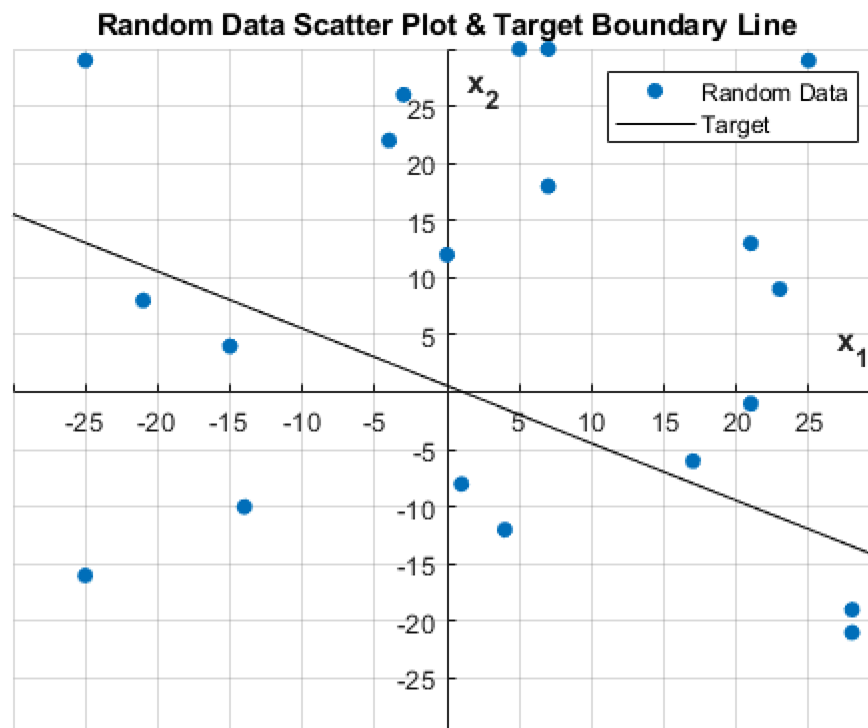
Part C. Plotting the Target Function

Instructions: Choose the line x1 + 2*x2 - 1.1 = 0 as your target function where the points on one side of the line map to: y = +1 (f = x1 + 2*x2 - 1.1 > 0) and the other points map to: y = -1 (f = x1 + 2*x2 - 1.1 < 0). Now, you have a set of 20 data points (X,y) as your separable data points.

Implementation: The given target function is solved for x2 (y) so that it can be plotted on the same plot with the data points generated in part B. The mapping to +1 and -1 for points above and below the line

is implemented in part D along with the color coding of the points for convenience and code readability. This way, the points are both color coded and mapped in a single loop rather than writing one loop to map them and another to color code them.

```
x1plot = -30:0.05:30; % Creates a vector of plot points in the x
 direction
x2plot = -0.5*x1plot + 0.55; % Plot target function
Plot1 = plot(x1plot, x2plot, 'DisplayName','Target','color','black');
legend % Add legend to plot of target function

hold off
```



Part D. Linear Separation and Plotting of Data

Instructions: Plot the points on the 2D plane labeling them with "+" or "-" or Red and Blue.

Implementation: A "for" loop is used to cycle through each row in the X matrix (or in other words, each of the 20 data points) to check if a point is above or below the target function line. If a point is above the line, it is mapped to a value of 1 and given a red fill. If a point is below the line, it is mapped to a value of -1 and given a blue fill. The mapping of the point is stored as a third column in the data matrix so that each point has an associated mapping in the matrix. The points are plotted one by one, red or blue, above or below the line by the "for" loop.

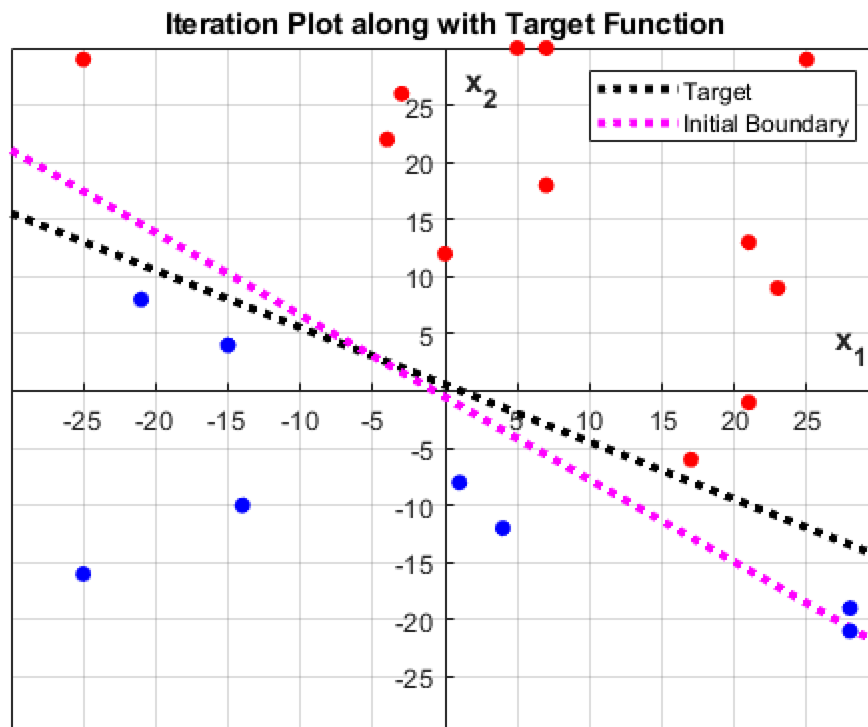
```
[nRowsInX, nColumnsInX] = size(X); % Store size of X matrix for
 looping
wTarg = [-1.1 1 2]; % target function weights = [w0, w1, w2]
```

```
figure('Name', 'HW2: Perceptron Plot 2')

Plot2 = plot(x1plot,
 x2plot,':','DisplayName','Target','color','black','LineWidth',3);
hold on % plot target function, separated data, and initial boundary
        % all in same figure


wVP = 0.002 + rand(1,3); % create and populate vector for random
 initial
                         % weight values. 0.001 offset is introduced
 to
                         % avoid a situation where the randomly
 generated
                         % weight value is 0 and one of the functions
 tries
                         % to then divide by 0, causing problems.

disp(" ")
disp("Weight Vector Initial Values: ")
disp(wVP)
disp(" ")

% Plot initial boundary line using randomly generated weight vector
 wVP
x1plotP = -30:0.05:30;
x2plotP = (-1*(wVP(2)/wVP(3))*x1plotP)-(wVP(1)/wVP(3));
Plot3 = plot(x1plotP, x2plotP,':','DisplayName','Initial
 Boundary','color','magenta','LineWidth',3);

redMatrix = zeros(2,2);
blueMatrix = zeros(2,2);
redCount = 1;
blueCount = 1;

% Generate linearly separable data separable by the given target
 function
for i = 1:nRowsInX % Loop through every row of the X matrix
    x1Loop = X(i,1); % Get the x1 value of the ith row
    x2Loop = X(i,2); % Get the x2 value of the ith row
    f = wTarg(1) + wTarg(2)*x1Loop + wTarg(3)*x2Loop; % Obtain f value
    if f > 0
        X(i,3) = 1; % assign a 1 to points above the line
        c = 'red';  % give points above the line a red color fill
        redMatrix(redCount,1) = X(i,1); % populate red matrix for
 later plotting
        redMatrix(redCount,2) = X(i,2);
        redCount = redCount + 1;
    else
        X(i,3) = -1; % assign a -1 to point below the line
        c = 'blue';  % give points below the line a blue color fill
        blueMatrix(blueCount,1) = X(i,1); % populate blue matrix for
 later plotting
```

```
            blueMatrix(blueCount,2) = X(i,2);
            blueCount = blueCount + 1;
        end
        % Plot and axis settings, limits, labels for plot 2
        title('Iteration Plot along with Target Function');
        xlabel('x_{1}', 'FontSize', 12, 'FontWeight', 'bold');
        ylabel('x_{2}', 'FontSize', 12, 'FontWeight', 'bold');
        ax = gca;
        ax.XAxisLocation = 'origin';
        ax.YAxisLocation = 'origin';
        xlim([-30 30]);
        xticks(-30:5:30);
        ylim([-30 30]);
        yticks(-30:5:30);
        grid on
        scatter(X(i,1),X(i,2),c,'filled');
        legend([Plot2,Plot3]);
    end

    % disp("Values in X matrix: ")
    % disp(X)
```

Weight Vector Initial Values:

0.4860 0.6410 0.8896



Part E. Implementing Perceptron Algorithm

Instructions: Implement and run the perceptron algorithm. You must write the perceptron code from scratch as opposed to using the code in software packages. Make sure to include declarations next to code lines for ease of readability.

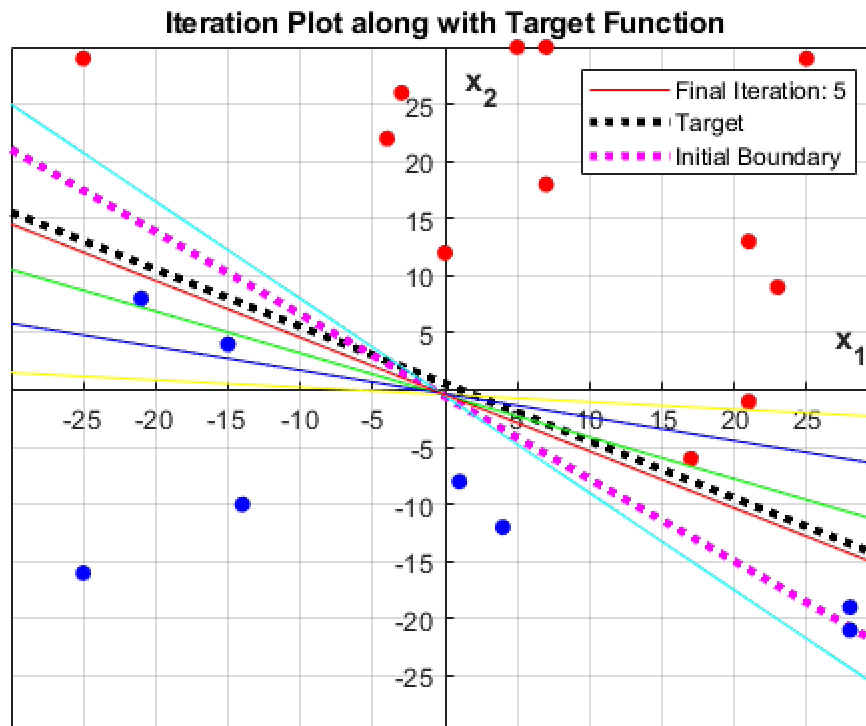
Implementation: A copy of the data matrix is generated and the new randomly weighted initial boundary line is used to determine a new mapping of the data points. The new mapping (the third column of the comparison matrix) and the initial mapping (the third column of the data matrix) are compared. If the new mapping and initial mapping of a given point are equal, then that point is not misplaced. If these mappings are not equal, then the point is misplaced. This data (misplaced, 1, or not misplaced, 0) is stored in a fourth column of the data matrix.

A "for" loop is then used to cycle through the matrix and find the first point which is labeled as misplaced. A function was written to update the weight values each time that a misplaced point is detected. After the update, the values in the data matrix and comparison matrix are regenerated and a new list of misplaced points is created. This process repeats until there are 0 misplaced points detected at which point the program is done and the final weights define a valid boundary line.

```
XCompare = X;     % Set comparison matrix equal to reference matrix X
isDone = false;   % initialize stop variable for while loop
colorIndex = 0;   % initialize color index for plotting iterations
loopCounter = 1;  % initialize loop counter for legend
nRate = 0.02;     % Set learning rate for weight update function

while isDone == false

    for i = 1:nRowsInX % Loop through every row of the X matrix
        x1Loop = X(i,1); % Get the x1 value of the ith row
 (coordinate)
        x2Loop = X(i,2); % Get the x2 value of the ith row
 (coordinate)
        f = wVP(1) + wVP(2)*x1Loop + wVP(3)*x2Loop; % Obtain new f
 value
        if f > 0
            XCompare(i,3) = 1; % assign a 1 to points above the line
        else
            XCompare(i,3) = -1; % assign a -1 to points below the line
        end
    end

%      disp("Values in Comparison matrix: ")
%      disp(XCompare)

    % Find number of misplaced points and mark their status in column
 4
    numPts = 0;

    for i = 1:nRowsInX
        if X(i,3) ~= XCompare(i,3) % if third column values are not
 equal
            numPts = numPts + 1;    % then increment the misplaced
 counter
```

```
            X(i,4) = 1;            % and set the fourth column value
 to 1
            XCompare(i,4) = 1;
        else
            X(i,4) = 0; % otherwise, set fourth column value to 0
            XCompare(i,4) = 0;
        end
    end


    % Exit the while loop and stop iterating when there are no more
    % misplaced points detected, but keep iterating if there are still
    % misplaced points.
    if numPts == 0
        isDone = true;
        x2plotPnew = (-1*(wVP(2)/wVP(3))*x1plotP)-(wVP(1)/wVP(3));
%          disp("X matrix misplaced pts: ")
%          disp(X)

%          fprintf('Misplaced Pts: %d\n', numPts)
    else
%          disp("X matrix misplaced pts: ")
%          disp(X)

%          fprintf('Misplaced Pts: %d\n', numPts)

        % Traverse matrix for misplaced points to adjust weights
        for i = 1:nRowsInX
            if X(i,4) == 1 % enter this code body if the point is
 misplaced
                x1Misp = X(i,1); % get x1 value of the misplaced point
                x2Misp = X(i,2); % get x2 value of the misplaced point
                wVP =
 UpdateWeights(wVP(1),wVP(2),wVP(3),x1Misp,x2Misp,nRate,X(i,3));
                dispTxt = ['Iteration: ',num2str(loopCounter)];
                disp(dispTxt)
                disp("Updated Weights: ")
                disp(wVP)
                disp(" ")
                % plot a new line for each iteration to track changes
                x2plotPnew = (-1*(wVP(2)/wVP(3))*x1plotP)-(wVP(1)/
wVP(3));
                plotColor = ChangeColor(colorIndex); % change line
 color
                legendTxt = ['Final Iteration:
 ',num2str(loopCounter)];
                plotFinal =
 plot(x1plotP,x2plotPnew,'DisplayName',legendTxt,'color',plotColor);
                legend([plotFinal,Plot2,Plot3]);
                if loopCounter == 1
                    iteration1 = x2plotPnew;
                elseif loopCounter == 2
                    iteration2 = x2plotPnew;
                elseif loopCounter == 3
```

```
                    iteration3 = x2plotPnew;
                end
                break
            end
        end
    end

    % increment the color index and loop counter after each iteration
    colorIndex = colorIndex + 1;
    loopCounter = loopCounter + 1;

end

iterationFinal = x2plotPnew;

disp("Number of iterations taken: ")
numIterations = loopCounter - 2;
disp(numIterations)

disp(" ")
disp("Final Weights: ")
disp(wVP)

% legend([Plot2,Plot3]); % add a legend for the lines
hold off
```



Questions Regarding Homework Assignment

How many iterations does it take to arrive at the solution boundary (estimated target function)? Plot 4 iterations including the final one showing the boundary line at each iteration. Draw function f line on the last plot and explain why they are different.

Average Number of Iterations Taken for Random Data (nRate = 0.02)

Run 1: 4

Run 2: 1

Run 3: 2

Run 4: 4

Run 5: 2

Run 6: 2

Run 7: 2

Run 8: 7

Run 9: 3

Run 10: 2

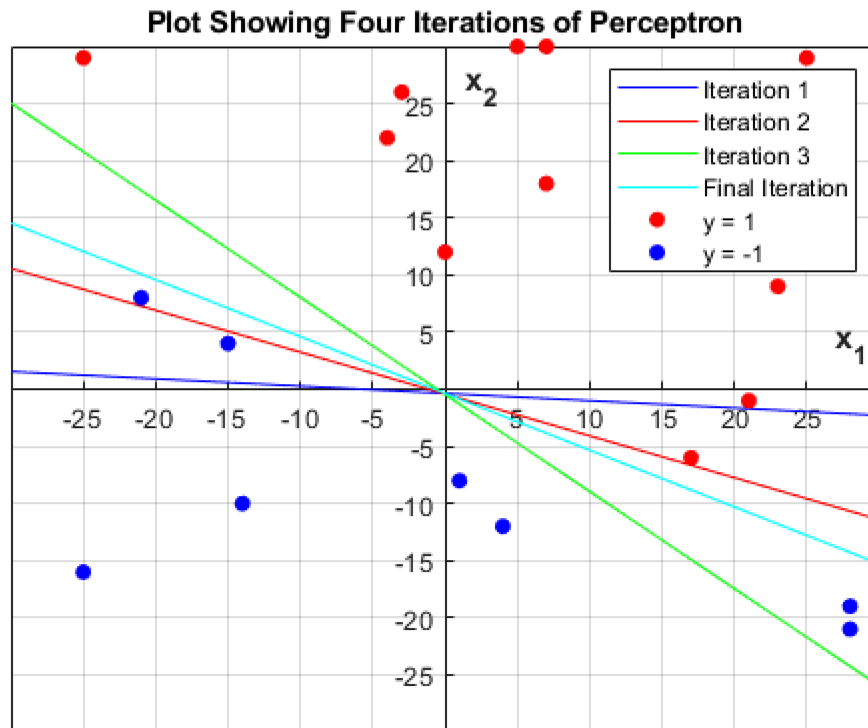
Average Number of Iterations Taken = 2.9

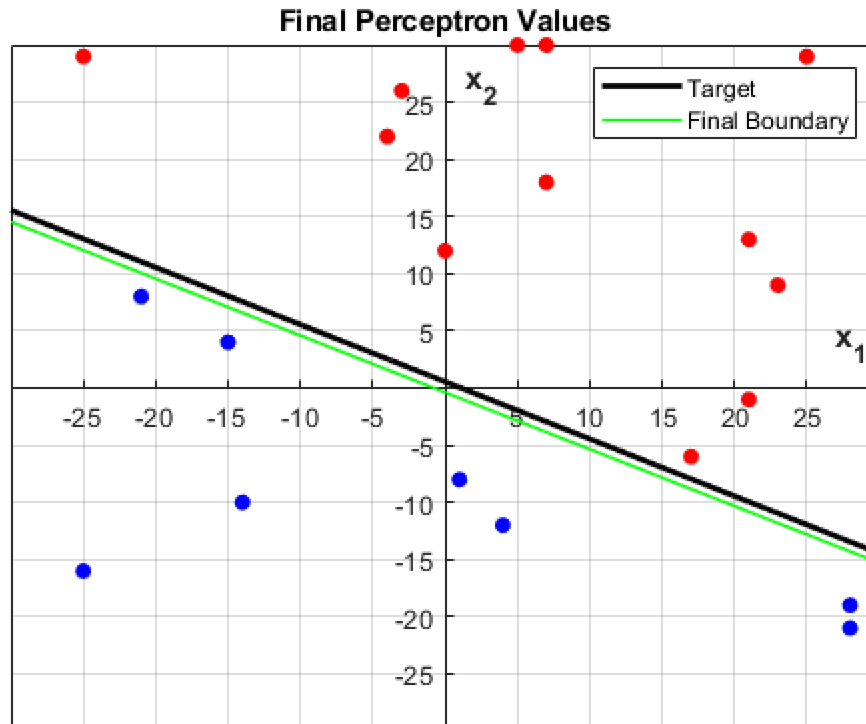
The target function line and the final weighted line found by the perceptron algorithm are different because there are infinitely many solutions which can be found by the perceptron algorithm as lines that "fit" or properly separate the data. The perceptron algorithm is simply checking for misplaced points and stops iterating when there are no more points misplaced, so it stops at the first solution. Given that we have only 20 data points, there is a higher likelihood that we have a larger range of lines which will separate the data. However, if we were to run the algorithm with more data, we would expect to see a final boundary line much closer to the target function because there are more data points, and therefore a lower likelihood that we have a large range of lines which will properly separate the data.

```
% Plot showing 4 iterations
if numIterations > 3
    figure('Name', 'HW2: Perceptron Plot 3')
    plot(x1plotP,iteration1,'blue');
    hold on
    plot(x1plotP,iteration2,'red');
    plot(x1plotP,iteration3,'green');
    plot(x1plotP,iterationFinal,'cyan');
    scatter(redMatrix(:,1),redMatrix(:,2),'red','filled');
    scatter(blueMatrix(:,1),blueMatrix(:,2),'blue','filled');
    title('Plot Showing Four Iterations of Perceptron');
    xlabel('x_{1}', 'FontSize', 12, 'FontWeight', 'bold');
    ylabel('x_{2}', 'FontSize', 12, 'FontWeight', 'bold');
    ax = gca;
```

```
    ax.XAxisLocation = 'origin';
    ax.YAxisLocation = 'origin';
    xlim([-30 30]);
    xticks(-30:5:30);
    ylim([-30 30]);
    yticks(-30:5:30);
    legend('Iteration 1','Iteration 2','Iteration 3','Final
 Iteration', 'y = 1', 'y = -1');
    grid on
    hold off
end


% Final plot showing final line, target line, and data points.

figure('Name', 'HW2: Perceptron Final Plot')

% Plot the target function again on the final plot for comparison
p1_final = plot(x1plot,
 x2plot,'DisplayName','Target','color','black','LineWidth',2);
hold on

% Plot the final boundary line in the same figure
x1plotP_final = -30:0.05:30;
x2plotP_final = (-1*(wVP(2)/wVP(3))*x1plotP_final)-(wVP(1)/wVP(3));

p2_final = plot(x1plotP, x2plotPnew,'DisplayName','Final
 Boundary','color','green','LineWidth',1);

% Plot the scatter plot (data points) on the same figure with the
 lines
for i = 1:nRowsInX % Loop through every row of the X matrix
    x1Loop = X(i,1); % Get the x1 value of the ith row
    x2Loop = X(i,2); % Get the x2 value of the ith row
    f2 = wVP(1) + wVP(2)*x1Loop + wVP(3)*x2Loop;
    if f2 > 0
        c = 'red';
    else
        c = 'blue';
    end
    % Plot and axis settings, limits, labels for final plot
    title('Final Perceptron Values');
    xlabel('x_{1}', 'FontSize', 12, 'FontWeight', 'bold');
    ylabel('x_{2}', 'FontSize', 12, 'FontWeight', 'bold');
    ax = gca;
    ax.XAxisLocation = 'origin';
    ax.YAxisLocation = 'origin';
    xlim([-30 30]);
    xticks(-30:5:30);
    ylim([-30 30]);
    yticks(-30:5:30);
    grid on
    scatter(X(i,1),X(i,2),c,'filled');
end
```

```
legend([p1_final,p2_final]);

hold off
```





Functions

```
% Function that takes in the weights, learning rate, and input and
% outputs new weights to update the boundary line
function wVP_new = UpdateWeights(w0, w1, w2, x1, x2, nRate, d)
    wVP_new(1) = w0 + (nRate * d * 1);
    wVP_new(2) = w1 + (nRate * d * x1);
    wVP_new(3) = w2 + (nRate * d * x2);
end

% Function that takes in the color index variable and outputs a
 different
% color for each iteration of the while loop
function color = ChangeColor(i)
    inew = mod(i,7);
    switch inew
        case 0
            color = 'yellow';
        case 1
            color = 'green';
        case 2
            color = 'cyan';
        case 3
            color = 'blue';
        case 4
```

```
            color = 'red';
        case 5
            color = 'black';
        case 6
            color = 'magenta';
    end
end
```

```
Iteration: 1
Updated Weights:
    0.4660    0.0810    1.2696

Iteration: 2
Updated Weights:
    0.4860    0.4210    1.1496

Iteration: 3
Updated Weights:
    0.4660    0.8410    0.9896

Iteration: 4
Updated Weights:
    0.4460    0.2810    1.3696

Iteration: 5
Updated Weights:
    0.4660    0.6210    1.2496

Number of iterations taken:
     5

Final Weights:
    0.4660    0.6210    1.2496
```

Final Comments & Conclusions

The average number of iterations required was calculated based on random data and a fixed learning rate. In this case, the learning rate was fixed at 0.02. Different sizes of data matrices (i.e. more points) and different learning rates were tested, and it was determined that changing the learning rate can improve the number of iterations taken in some cases, but not in others. This because the learning rate directly influences the updated weights proportionally. If we have a larger learning rate, the updated weights will change more drastically. A smaller learning rate causes smaller changes in the updated weights.

In this program, the data is generated randomly on every single run, so no two data sets are identical (ideally, unless by some insane odds, two randomly generated sets wind up identical). Likewise, the weights for the initial boundary line are randomly generated on each run, so the boundary line never starts in the same place. The result of this randomness is that the number of iterations changes with each run. Most of the time, the number of iterations is below ten. However, there are some cases where two data points (one red and one blue) are on either side of the target line and very very close to it, resulting in an enormous number of iterations required for the algorithm to find the correct line. This is because the window is so small for possible solutions that the algorithm has to bounce back and forth around the solution until it finally settles very close to the target line.

James Skelly, 2021

Published with MATLAB® R2018b