Assignment #1

See the bottom of the document for attached functions. Run `PartitionTest.m` to duplicate the results.

Problem #1

Take the third attribute (column #4 of the original data file). Divide the value range into four intervals of equal width. What are the ranges for each interval?

See `EqWidthPartition()` for reference

[0.0000, 1.1225)
[1.1225, 2.2450)
[2.2450, 3.3675)
[3.3675, 4.4900]

Problem #2

Find the Gini index, information gain, and gain ratio for each of the three splitting points. (Four intervals have three splitting points at their boundaries).

See `SplitMetrics()` for reference

gini_idx =

0.6601 [First split]

0.6402 [Second split]

0.6535 [Third split]

info_gain =

0.2376 [First split]

0.3409 [Second split]

0.3205 [Third split]

gain_ratio =

0.4619 [First split]

0.5881 [Second split]

0.4784 [Third split]

Problem #3

Split the value range into four intervals of equal frequency. What are the ranges for each interval?

Note: for equal intervals that would result in splitting equal data points, the equal data points were chosen to all be included in the *first* interval.

[0.0000, 2.1900)
[2.1900, 3.4800)
[3.4800, 3.6100)
[3.6100, 4.4900]

Problem #4

Find the GINI index, information gain, and gain ratio for each of the three splitting points.

gini_idx =

0. 6457 [First split]

0. 6890 [Second split]

0. 7219 [Third split]

info_gain =

0. 3174 [First split]

0. 1971 [Second split]

0. 0735 [Third split]

gain_ratio =

0. 5569 [First split]

0. 2846 [Second split]

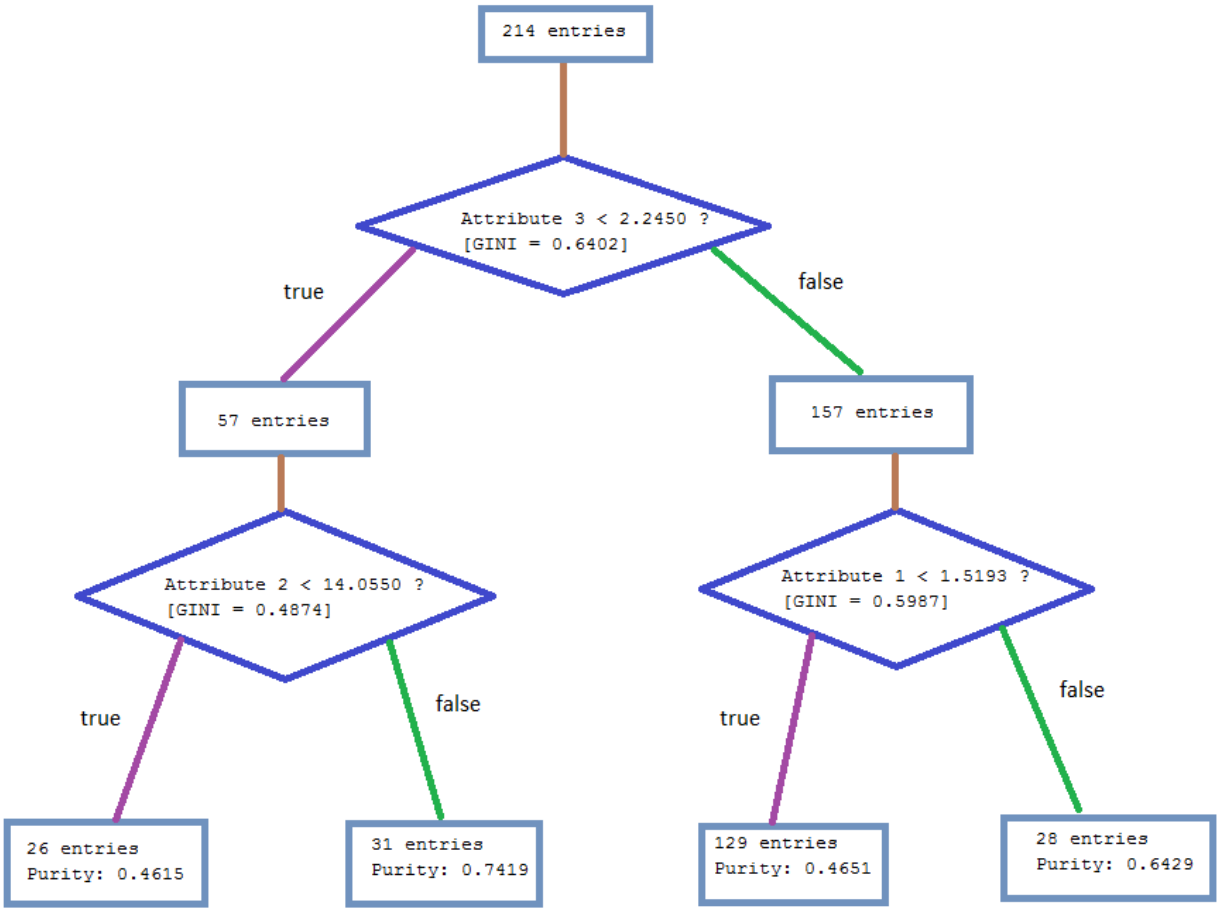
0. 1365 [Third split]

Problem #5 and Problem #6

5. Consider only the first three attributes. Perform equal width partitions for each of these three attributes. From among the nine splitting points select the best attribute for the decision tree using GINI index as the metric. Now repeat the process for the dataset at each branch of the resulting decision node. This will give you a two level decision tree consisting of three question/test nodes.

6. What is the class purity for each of the four resulting leaf nodes of the decision tree constructed in #5 above?

The solutions to these problems are shown in the below tree (purities shown in leaf nodes):



PartitionTest.m

```
% Hw1, #1-4
% Garrett Scholtes
% Compute answers for these problems

clear all;
clc;

data = xlsread('glassdataB.xls');

% Problem #1
% `bounds` tells you the interval boundaries
% `counts` tells you how many data are in each interval
[sorted_data, bounds_width, class_counts] = EqWidthPartition(data, 4, 4, 11);
fprintf('Problem 1:');
display(bounds_width);

% Problem #2
% Each variable is an array containing the indexes for each of the splits
[gini_idx, info_gain, gain_ratio] = SplitMetrics(class_counts, size(data,1));
fprintf('\n\nProblem 2:');
display(gini_idx);
display(info_gain);
display(gain_ratio);

% Problem #3
% Like the above problem #1, each variable is an array containing the
% indexes for each of the splits (but this time, equal frequency not width)
[sorted_data, bounds_freq, class_counts_freq] = EqFreqPartition(data, 4, 4, 11);
fprintf('\n\nProblem 3:');
display(bounds_freq);

% Problem #4
% Each variable is an array containing the indexes for each of the splits
[gini_idx_freq, info_gain_freq, gain_ratio_freq] = SplitMetrics(class_counts_freq, size(data,1));
fprintf('\n\nProblem 4:');
display(gini_idx_freq);
display(info_gain_freq);
display(gain_ratio_freq);

% Problem #5
% Perform equal width partitioning for each of the first 3 attributes.
% Use the gini index in order to choose the best splits to form a
% complete decision tree with depth 2
[split1, column1, gini1, left_data1, right_data1] = BuildTree( data, 4, 2:4 );
% Now build the child nodes from the resultant parent
[split1a, column1a, gini1a, left_data1a, right_data1a] = BuildTree( left_data1, 4, 2:4 );
[split1b, column1b, gini1b, left_data1b, right_data1b] = BuildTree( right_data1, 4, 2:4 );

% Problem #6
% Compute the purity for each child node in the above decision tree
[purity_l1a, class_l1a] = Purity(left_data1a, 11);
```

```
[purity_l1b, class_l1b] = Purity(left_data1b, 11);
[purity_r1a, class_r1a] = Purity(right_data1a, 11);
[purity_r1b, class_r1b] = Purity(right_data1b, 11);

% ------ %

fprintf('\n\nProblems 5 and 6:\n');
fprintf('\nRoot question:\nAttribute %d < %0.4f ?\n[GINI = %0.4f]\n', column1-1, split1, gini1);
fprintf('\nLeft question:\nAttribute %d < %0.4f ?\n[GINI = %0.4f]\n', column1a-1, split1a,
gini1a);
fprintf('\nRight question:\nAttribute %d < %0.4f ?\n[GINI = %0.4f]\n', column1b-1, split1b,
gini1b);

fprintf('\nRoot state ----------------:\n%d entries\n', size(data, 1));

fprintf('\nRoot question - left answer:\n%d entries\n', size(left_data1,1));
fprintf('\nRoot question - right answer:\n%d entries\n', size(right_data1,1));

fprintf('\nLeft question - left answer:\n%d entries\nPurity: %0.4f\n', size(left_data1a,1),
purity_l1a);
fprintf('\nLeft question - right answer:\n%d entries\nPurity: %0.4f\n', size(right_data1a,1),
purity_r1a);
fprintf('\nRight question - left answer:\n%d entries\nPurity: %0.4f\n', size(left_data1b,1),
purity_l1b);
fprintf('\nRight question - right answer:\n%d entries\nPurity: %0.4f\n', size(right_data1b,1),
purity_r1b);
```

Published with MATLAB® R2013a

EqWidthPartition.m

```
function [ sorted_matrix, bounds, class_counts ] = EqWidthPartition( matrix, n, col, class_col )
%EqWidthPartition - split matrix into cell partitions
%   `matrix` is sorted by data in column `col` and then split into
%      `n` equal width partitions. `class_col` is the index of the column
%       in `matrix` which contains class labels
%    Return value is
%      `sorted_matrix`, the data matrix, `matrix`, sorted by column `col`
%      `bounds`, an array with the `n-1` splits for the `n` partitions
%      `class_counts`, like counts, but per class (each slice in the third
%           dimension is like `counts`, but only for a certain class)

    sorted_matrix = sortrows(matrix, col);
    attrs = sorted_matrix(:,col);
    classes = sorted_matrix(:,class_col);
    unique_classes = unique(classes);

    lower_bound = min(attrs);
    upper_bound = max(attrs);

    bounds = linspace(lower_bound, upper_bound, n+1);
    bounds = bounds(2:end-1)'; % trim min and max - they are not splits
```

```
    %counts = zeros(n-1,2);
    %entries = size(attrs, 1);
    %for k = 1:n-1
    %    counts(k,1) = sum(attrs < bounds(k));
    %    counts(k,2) = entries - counts(k,1);
    %end

    number_of_classes = size(unique_classes,1);
    class_counts = zeros(n-1,2, number_of_classes);
    for c = 1:number_of_classes
        class = unique_classes(c);
        for k = 1:n-1
            class_counts(k,1,c) = sum(attrs <= bounds(k) & classes == class);
            class_counts(k,2,c) = sum(attrs > bounds(k) & classes == class);
        end
    end

    bounds = [lower_bound; bounds; upper_bound];

end
```

Published with MATLAB® R2013a

SplitMetrics.m

```
function [ gini_idx, info_gain, gain_ratio ] = SplitMetrics( class_counts, records )
%SplitMetrics
%   Given `class_counts` that come from `EqWidthPartition()` and the number
%   of records `records` from the dataset, compute these indexes for
%   various splits

    counts = sum(class_counts(:,:,:), 3);
    number_of_classes = size(class_counts,3);

    num_partitions = size(class_counts, 1);

    gini_idx = zeros(num_partitions,1);
    info_gain = zeros(num_partitions,1);
    gain_ratio = zeros(num_partitions,1);

    % Root entropy
    root_entropy = 0;
    for k = 1:number_of_classes
        probability = class_counts(1, 1, k) + class_counts(1, 2, k);
        probability  = probability / records;
        if probability > 0
            root_entropy = root_entropy - probability*log(probability);
        end
    end

    for split = 1:num_partitions
```

```
        gini_idx(split) = 0;
        info_gain(split) = root_entropy;
        gain_ratio(split) = 0;

        for branch = [1 2]
            gini = 0;
            entropy = 0;

            for class = 1:number_of_classes
                probability = class_counts(split, branch, class)/counts(split, branch);
                gini = gini + (probability)^2;
                if probability > 0
                    entropy = entropy - probability*log(probability);
                end
            end
            gini = 1-gini;
            density = counts(split, branch) / records;

            gini_idx(split) = gini_idx(split) + gini * density;
            info_gain(split) = info_gain(split) - entropy * density;
            gain_ratio(split) = gain_ratio(split) - density * log(density);
        end
        gain_ratio(split) = info_gain(split) / gain_ratio(split);
    end

end
```

Published with MATLAB® R2013a

EqFreqPartition.m

```
function [ sorted_matrix, bounds, class_counts ] = EqFreqPartition( matrix, n, col, class_col )
%EqWidthPartition - split matrix into cell partitions
%   `matrix` is sorted by data in column `col` and then split into
%      `n` equal width partitions. `class_col` is the index of the column
%       in `matrix` which contains class labels
%    Return value is
%      `sorted_matrix`, the data matrix, `matrix`, sorted by column `col`
%      `bounds`, an array with the `n-1` splits for the `n` partitions
%      `class_counts`, like counts, but per class (each slice in the third
%           dimension is like `counts`, but only for a certain class)

    sorted_matrix = sortrows(matrix, col);
    attrs = sorted_matrix(:,col);
    classes = sorted_matrix(:,class_col);
    unique_classes = unique(classes);

    lower_bound = min(attrs);
    upper_bound = max(attrs);

    num_entries = size(matrix,1);
    bounds = linspace(1, num_entries+1, n+1);
```

```
    bounds = round(bounds(2:end-1)); % trim end and front index
    bounds = attrs(bounds); % Use values, not indexes for below logic

    %counts = zeros(n-1,2);
    %entries = size(attrs, 1);
    %for k = 1:n-1
    %    counts(k,1) = sum(attrs < bounds(k));
    %    counts(k,2) = entries - counts(k,1);
    %end

    number_of_classes = size(unique_classes,1);
    class_counts = zeros(n-1,2, number_of_classes);
    for c = 1:number_of_classes
        class = unique_classes(c);
        for k = 1:n-1
            class_counts(k,1,c) = sum(attrs <= bounds(k) & classes == class);
            class_counts(k,2,c) = sum(attrs > bounds(k) & classes == class);
        end
    end

    bounds = [lower_bound; bounds; upper_bound];

end
```

Published with MATLAB® R2013a

BuildTree.m

```
function [ threshold, column, gini, left_data, right_data ] = BuildTree( matrix, n, cols )
%BuildTree
%   This should really be "BuildNode" - this only does one node at a time,
%   and the other 2 nodes will just be hard coded for simplicity of the
%   problem
%   `matrix` - the dataset
%   `cols`   - an array specifying the index of the columns to be analyzed
%       (i.e., which attributes to analyze)
%   `n`      - number of partitions to test
%   Return value
%   `threshold`  - the value of the split threshold
%   `column` - which column (attribute) is being used to split
%   `gini`   - the gini index for the chosen split
%   `left_data` - a new data matrix, filtered to only contain entries whose
%        selected attribute is below the split threshold
%   `right_data` - a new data matrix, with only those above threshold

    gini = 1; % higher than the worst case -- we will now minimize this
    threshold = 0; % Will set this based on best Gini
    sorted_by_best = []; % Will be set to data sorted by the optimal chosen attribute

    for col = cols
            [sorted_data, bounds, class_counts] = EqWidthPartition(matrix, n, col, 11);
            [gini_idx, ~, ~] = SplitMetrics(class_counts, size(matrix,1));
```

```
            for k = 1:n-1
                if gini_idx(k) < gini
                    gini = gini_idx(k);
                    threshold = bounds(k+1);
                    column = col;
                    sorted_by_best = sorted_data;
                end
            end
    end
    left_predicate = sorted_by_best(:,column) <= threshold;
    right_predicate = ones(size(left_predicate,1),1) - left_predicate;

    left_data = sorted_by_best(logical(left_predicate),:);
    right_data = sorted_by_best(logical(right_predicate),:);
end
```

Published with MATLAB® R2013a

Purity.m

```
function [ max_purity, max_class ] = Purity( matrix, class_col )
%Compute purity of a decision tree leaf node

    max_purity = 0;
    class_mode = 0;

    classes = matrix(:,class_col);
    unique_classes = unique(classes);
    num_entries = size(matrix,1);

    for class_ = unique_classes'
        total = sum(classes == class_);
        if total > class_mode
            max_purity = total / num_entries;
            max_class = class_;
            class_mode = total;
        end
    end

end
```

Published with MATLAB® R2013a