Feedback sheet: Assignment

This feedback sheet is meant to help you to improve your assignment reports, and to understand the grading criteria.

Part of the work of an applied mathematician is to communicate with experts that may not be mathematicians. Therefore, it is important to develop one's reporting skills. In this course, the quality of the reporting is an important element in grading.

Below, a list of important points concerning the reporting are given, and it also serves as a rubric for feedback.

1. The report should be self-explanatory, not an answer to a question. This means that you have to explain what the problem is that you are addressing. Preferably, do not just copy the assignment but explain in your own words what the assignment asks you to do. **A printed code is not a sufficient answer!**

   Your grade:

   ☐ Excellent: No need to improve
   ☐ Good: Acceptable, leaving space for improvement
   ☐ Below standard : Requires more work

2. The report should be carefully typeset, paying attention to the layout. Sloppy reporting gives a negative impression and raises questions about the correctness of the results, so one needs to pay attention on the visual quality.

   Your grade:

   ☐ Excellent: No need to improve
   ☐ Good: Acceptable, leaving space for improvement
   ☐ Below standard : Requires more work

3. Well planned graphs and plots are an integral part of good reporting. When you include a plot in the report, pay attention to the following:

   (a) Make sure that the fonts are of readable size. The default font in Matlab is not acceptable, so increase the font size, e.g., using the command `set(gca,'FontSize',15)`.

   (b) Lines and curves need to be thick enough to be readable, and markers must be large enough.

   (c) Pay attention to axes. For instance, if you plot a circle, make sure that it does not appear as an ellipse, using commands like `axis('square')` or `axis('equal')`. Also, think about the scaling of the axis so that they convey the information you want. Sometimes a logarithmic scale is more informative.

   (d) Select the graphical output carefully. Is a histogram better than a continuous curve? Discrete quantities may be better represented by dots or bars than by continuous curves etc.

(e) Consider marking the axes with a label, using commands `xlabel, ylabel`.

Your grade:

- ☐ Excellent: No need to improve
- ☐ Good: Acceptable, leaving space for improvement
- ☐ Below standard : Requires more work

4. Technical correctness. Each assignment will be checked for correctness. If you get a grade less than excellent, go and check the comments on the report.

   Your grade:

   - ☐ Excellent: No need to improve
   - ☐ Good: Acceptable, leaving space for improvement
   - ☐ Below standard : Requires more work

5. Code: Including a code, or a piece of it is recommended when applicable. Pay attention on your code: A code with no comments is below standard. Well documented code is a great asset for yourself, too!

   Your grade:

   - ☐ Excellent: No need to improve
   - ☐ Good: Acceptable, leaving space for improvement
   - ☐ Below standard : Requires more work

Final Project: Tree Regression
MATH 444: Mathematics of Data Science

David Frost

Introduction: Tree Classifiers and Regression

In this analysis, we will use regression trees to recreate an image from just a tiny fraction of its pixels. We want to recreate images that have been subjected to extreme compression. First, we must clearly define what a regression tree is, and we will give a brief primer on trees for those who are unfamiliar.

A tree data structure consists of a node with values and pointers to children. In this case, we will use a binary tree, which has pointers to a left child and a right child. Each node in a regression tree contains the average red, green, and blue value of each of the pixels within the rectangle represented by this node. Additionally, each node contains the RGB values of the pixels that are in each node. When a node is split, it becomes a non-leaf node and all of its values are partitioned between its two children. This means that any non-leaf node (also referred to as an interior node) is not used for the final plotting of the image. Therefore, only the leaf nodes are of importance when using a regression tree.

A regression tree is a tree used to approximate a function. In the case of using a regression tree to approximate an image from a tiny fraction of its pixels, we first try to partition the image into rectangles whose colors are filled in using one of the kept pixels, but not necessarily the nearest pixel; rather, each rectangle's size and location is determined by minimizing vectorial mean square error during splitting to ensure optimal splitting of rectangles. Vectorial error is needed since RGB vectors have 3 values each, one for red, one for green, and one for blue. Initially, the entire image is approximated using one rectangle of the image's dimension with one color. Obviously, most images aren't just one solid-colored rectangle, so we want more accuracy than that. In the first splitting, we split the one rectangle into two rectangles, each of which has as its color the average RGB value of the collection of pixels within its area. We can keep doing this splitting until each rectangle has only one pixel within its area, such that for each pixel that we are given to reconstruct the image, we have one rectangle. This is referred to as the maximal tree, where each leaf is a pure leaf, which means that each leaf has only one pixel associated with it, and thus only one set of RGB values. A mixed (impure) leaf is one which has multiple pixels associated with it, and thus the RGB value of its associated rectangle is the mean of the RGB values of its enclosed pixels. A maximal tree might be too costly, and if the splitting is stopped before the maximal tree is achieved, then mixed leaves will have to be used in the regression.

In the case of this regression tree, we can think of the function we are approximating as a map from a set of x and y coordinates on the image to an RGB value. For example, if we have 15000 pixels with which to recreate the image, then the maximal tree would result in a piecewise function with 15000 different pieces.

Splitting rectangles optimally is crucial for properly recreating the image. Therefore, we must specify how to optimally split rectangles.

Optimal Splitting and Example

To optimally split a 2-D rectangle, we must consider splitting it along either the X or Y axes, and along those axes, we must consider every possible split. A possible split is one whose split index on a dimension is at the midpoint between two pixels in that rectangle. So for example, if we have two pixels, one with an x-value of 1 and another with an x-value of 2, one possible split index for the x axis would be 1.5.

Since there are so many ways to split a rectangle with many pixels, we are interested in how to *optimally* split a rectangle. In this case, we consider an optimal split to be one which minimizes the mean squared error. We calculate this by first finding $\vec{c_1}$ and $\vec{c_2}$, which are the averages of the RGB values of the pixels within Rectangle 1 and Rectangle 2 resulting from a possible split. $\vec{v^{(j)}}$ represents the RGB vector for the pixel at (x_j, y_j). Depending on whether the split is along the x-axis or the y-axis, we use one of the following two formulas:

$$F_x(s, \vec{c_1}, \vec{c_2}) = \sum_{x^{(j)} \leq s} ||\vec{v}^{(j)} - \vec{c_1}||^2 + \sum_{x^{(j)} > s} ||\vec{v}^{(j)} - \vec{c_2}||^2$$

$$F_y(s, \vec{c_1}, \vec{c_2}) = \sum_{y^{(j)} \leq s} ||\vec{v}^{(j)} - \vec{c_1}||^2 + \sum_{y^{(j)} > s} ||\vec{v}^{(j)} - \vec{c_2}||^2$$

We want to pick the splitting that minimizes its respective error equation, and that has a lower than or equal to error than any splitting along any other dimension as well. This is what we refer to as the optimal splitting.

The abstract description of optimal splitting of rectangles with RGB pixels can definitely be confusing, so here is a computed example to show what we are talking about. There are four possible splits here: between P1 and P3 along on y-axis, between P2 and P3 along the y-axis, between P1 and P2 on the x-axis, and between P2 and P3 on the x-axis. Note that for each of these splits, there are two pixels in one rectangle and one pixel in another rectangle, so only one rectangle will contribute to the error and the other rectangle's error will be zero.

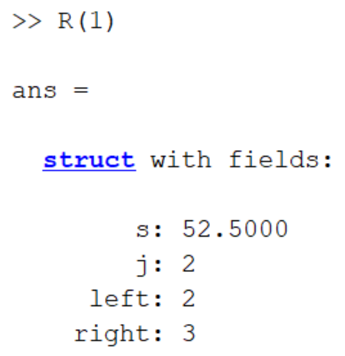
Possible split 1: We have P2 and P3 in the same rectangle on the right and P1 in its own rectangle. The average $\vec{c_2} = [0.65,\ 0.15,\ 0.15]$. The overall error is $||[0.4, 0.1, 0.3] - [0.65, 0.15, 0.15]||^2 + ||[0.9, 0.2, 0.0] - [0.65, 0.15, 0.15]||^2 = 0.175$.

Possible split 2: We have P1 and P3 in the same rectangle on the left and P2 in its own rectangle. The average $\vec{c_1} = [0.5,\ 0.35,\ 0.1]$. The overall error is $||[0.1, 0.5, 0.2] - [0.5, 0.35, 0.1]||^2 + ||[0.9, 0.2, 0.0] - [0.5, 0.35, 0.1]||^2 = 0.385$.

Possible split 3: We have P! and P3 in the same rectangle on the top and P2 in its own rectangle at the bottom. Note that this will result in the same overall error of 0.385 as Possible split 2.

Possible split 4: We have P1 and P2 in the same rectangle on the bottom and P3 in its own rectangle. The average $\vec{c_1} = [0.25,\ 0.3,\ 0.25]$ is for the bottom rectangle. The overall error is $||[0.1, 0.5, 0.0] - [0.25, 0.3, 0.25]||^2 + ||[0.4, 0.1, 0.3] - [0.25, 0.3, 0.25]||^2$ The overall error is 0.13.

Therefore, we can see that the 4th possible split is the best one. The 4th split has j_opt = 2 (y-axis) and s_opt = 0.5250 (midpoint between P1 and P3 along y-axis). To confirm, we can see that the MATLAB code implementation got the same result. Please note that the x and y coordinates are all scaled up by a factor of 100 due to glitches with getting the *fill* command to work with $(x, y) \in [0, 1]$. See Figure 1 for this confirmation.



```
>> R(1)

ans =

  struct with fields:

        s: 52.5000
        j: 2
     left: 2
    right: 3
```

Figure 1: The MATLAB implementation of the split algorithm yields the correct answer. s-opt had to be scaled up by 100 due to MATLAB issues.

After the first split, there are two rectangles, one with one pixel and another with two pixels, so any split of the rectangle with two pixels will work, as they will all result in an error of 0 as every pixel will then be in its own rectangle. Therefore, we only need to test the first split. See Figure 2 to see the layout of the pixels and the results of the first split, as well as the final image from the maximal tree.

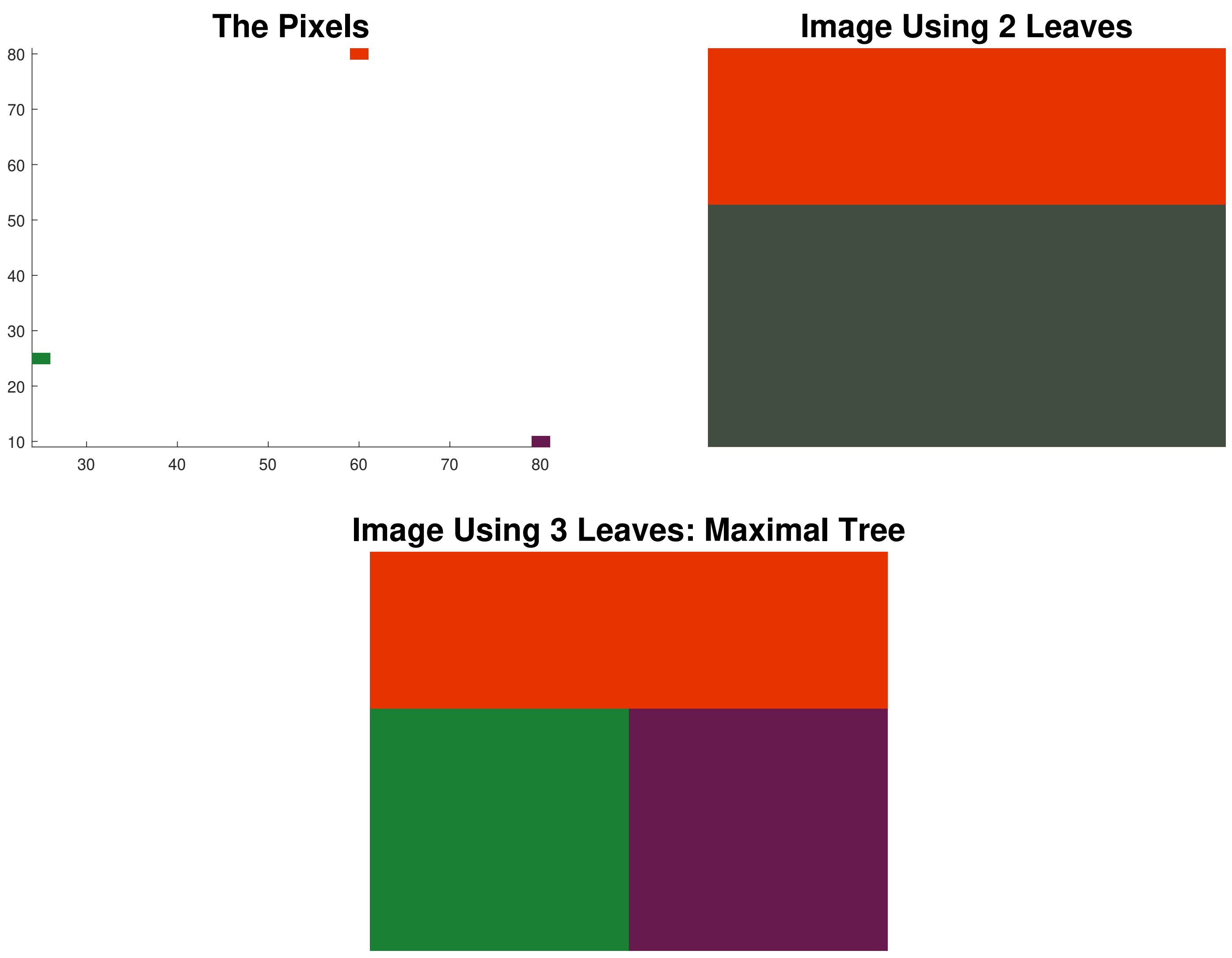


Figure 2: Left: The locations of the pixels in the image. Right: The image plotted after the first split. Lower: The image plotted from the maximal tree with 3 rectangles.

Implementation of Trees

To actually implement the tree regression algorithm, there are some generally good guidelines.

- Create a tree data structure using a Node class if possible in a language with objects. We implemented it in MATLAB and the nodes in the tree were structs.
- The fields/properties of the node objects should include pointers to left and right children, as well as the RGB values/pixels associated with the node. A convenient field would be average value of RGB values within the node for convenience/caching. Index value is important for determining what the index values of a split node's two children should be.

For expanding the regression tree using splitting, any tree traversal may be used, but using level-order traversal will result in the most "even" splitting. A tree traversal is an operation in which you visit each node of a tree once. The different types of tree traversals are:

- Pre-order: Explore root node, explore left subtree, explore right subtree.
- Level-order: Explore ith level of nodes before moving to explore the $(i + 1)$th level of nodes next. Level refers to the distance of each node from the root node.
- In-order: Explore left subtree, explore root node, explore right subtree.
- Post-order: Explore left subtree, explore right subtree, explore root node.

Note that all of these except for level-order will result in regression trees that are not maximal in having one side of an image with a great amount of detail and another side with little detail. Therefore, level-order traversal is the best tree traversal for building a regression tree assuming that equal detail is desired across the image, but for building a maximal tree, any traversal method will yield the same result.

Outline of Tree Regression Algorithm

We can now summarize the tree regression algorithm as follows:

1. Begin with sparse matrix representation of the few given pixels of the image. Initialize the root node to be the first leaf which contains every pixel. Initially, the only node in the pure leaf queue will be the root node.
2. Split oldest leaf node using optimal splitting and then partition its pixels between its two children. Update its pointers to its children and add its two children to one of the data structures for leaves, either the pure list or the mixed queue. Remove this leaf node from the pure leaf queue since it is no longer a leaf.
3. Repeat step 2 until either the number of splits or leaves is satisfactory or there are no mixed leaves anymore.
4. Iterate over the list of pure leaves and reconstruct the image by plotting rectangles based on the x and y values associated with each leaf node and the average RGB values of the pixels within that node.

Example: Mystery Image

In this analysis, we use the tree regression to analyze a mysterious image. We do not have any prior information about what this image represents. We are given 15000 pixels that are uniformly sampled without replacement from the mysterious image. We do know that the mysterious image has dimensions 2592×1456. It is given in sparse matrix form, so we have a *cols* vector of dimensions $\mathbb{Z}_+^{15000\times 1}$, a *rows* vector of dimensions $\mathbb{Z}_+^{15000\times 1}$, and a vals matrix of dimensions 15000×3, where $v_{ij} \in [0, 1]$. RGB values in many other contexts are $[0, 1, 2, ..., 255]$, but in this case, we normalize them to be between 0 and 1.

We don't know a lot about the original image, but we do know that its dimensions are 2592×1456, so the total number of pixels is 3,773,952. Therefore, the total amount of information we have about the image is

$$\tfrac{15000}{2592\times 1456} \times 100\% \approx 0.4\%$$

Having less than 1 in 200 pixels and trying to reconstruct an image would normally seem like an impossible task, as this is extreme compression. However, we ran the tree regression algorithm to see whether it could reconstruct the image. See Figure 3 for the images made from the early stages of the regression tree. See Figure 4 for the images made from the late stages of the regression tree. See Figure 5 for the image made from the maximal regression tree.

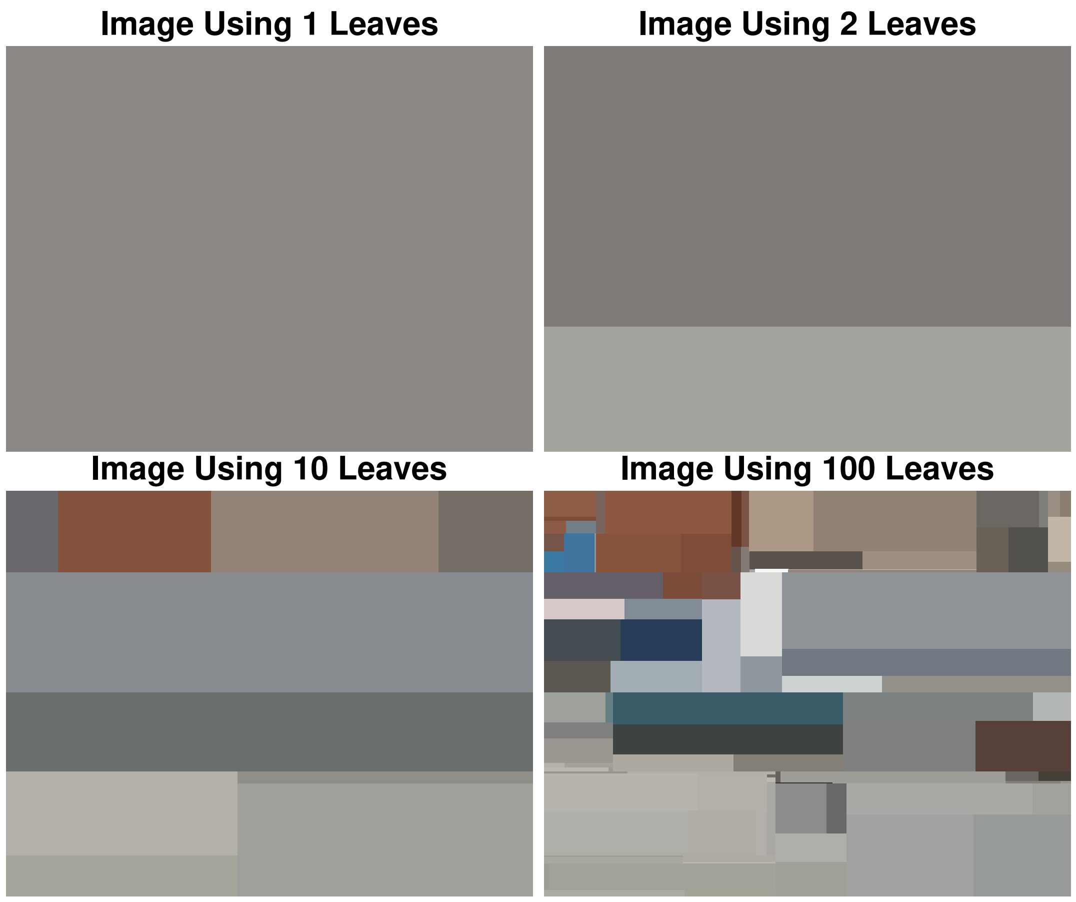


Figure 3: The very early stages of the images produced from the leaves of the tree regression. Even with 100 leaves, there is essentially no detail.

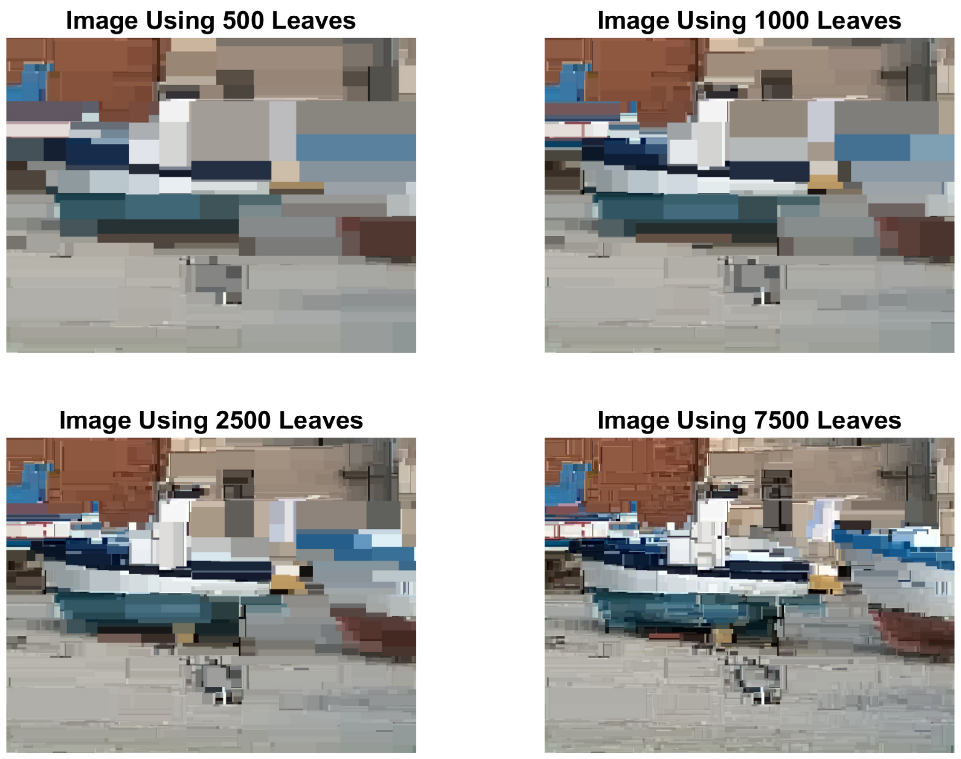


Figure 4: The intermediate and late stages of the images produced from the leaves of the tree regression. By the time we have 2500 leaves, we begin to get close to the amount of detail in the maximal tree.

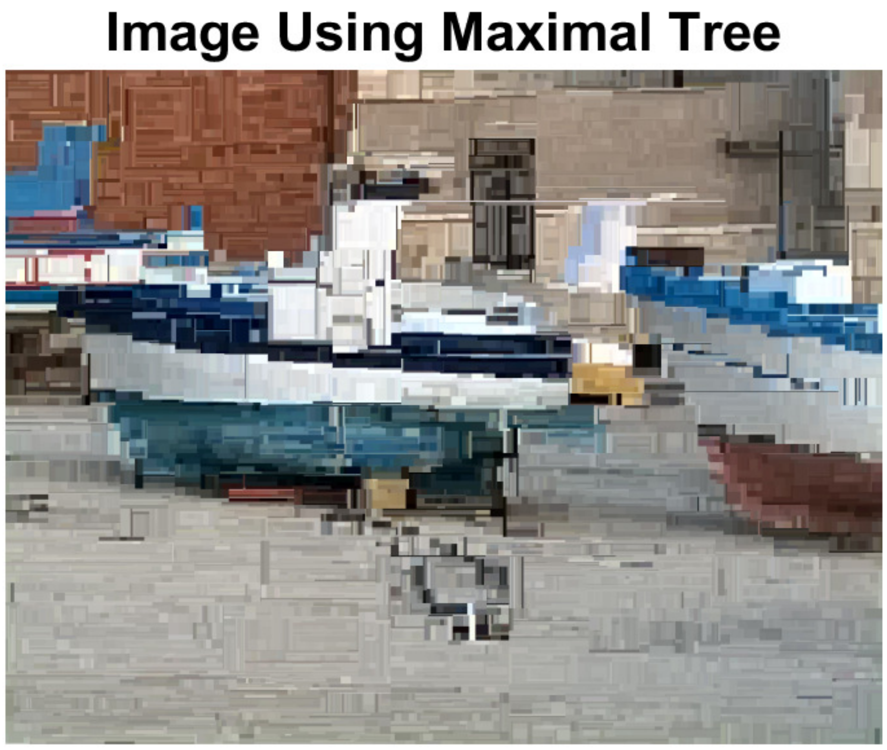


Figure 5: The most accurate image approximation from the maximal regression tree.

We're still not sure what this is; it could be a waterfall due to the pattern, or possibly boats? But the gray ground would not indicate boats, unless these boats are on a pile of rocks... We are leaning towards the waterfall, but the shape of some of the stuff in the middle actually looks possibly like sails on a boat.

Overall, the amount of detail is extraordinary given that only 0.4% of the pixels were given to us. It's incredible that we can even make reasonable guesses as to what the image represents since we have so little information given to us. Tree regression is a powerful method for artificial intelligence and machine learning and demonstrates a lot of promise.

Appendix

Code for main_script.m

```
% In this script, we use tree regression to recreate an image
% from just a small uniformly sampled subset of its pixels
load MysteryImage.mat
text_size = 20;
% Decide what the max number of splits is, maximal tree
% is probably too costly
max_splits = Inf;
% The only mixed leaf at the beginning is the root node
```

```
mixed_leaves = [1];
% Root node covers entire image
min_x = min(cols);
max_x = max(cols);
min_y = min(rows);
max_y = max(rows);
% Initialize the leaf of the tree
% The split index
R(1).s = [];
% The split dimension
R(1).j = [];
% The left child node index
R(1).left = [];
% The right child node index
R(1).right = [];
% The node index
R(1).I = 1;
% The x dimensions that this node has for the image
R(1).x = [min_x; max_x];
% The y dimensions that this node has for the image
R(1).y = [min_y; max_y];
% The y-axis values for the pixels in this node
R(1).rows = rows;
% The x-axis values for the pixels in this node
R(1).cols = cols;
% The RGB values for the pixels in this node
R(1).vals = vals;
% The average RGB value for the pixels in this node
R(1).val = mean(vals);
% Counter for keeping track of figure numbers
counter = 1;
% Index for the next leaf that is created
leaf_index = 2;
% List of indices of pure leaves
pure_leaves = [];
% List of indices of impure (mixed) leaves
impure_leaves = [1];
m = max(cols);
n = max(rows);
num_splits = 0;
% While loop that creates the tree. Will stop running when we either have
% the maximal tree or we have done the maximum number of splits
while ~isempty(impure_leaves) && (num_splits < max_splits)
    % Calculate number of leaves to determine whether to show intermediate
    % progress
    num_leaves = length(impure_leaves) + length(pure_leaves);
    % Show image at various stages
    % c1, c2, etc. = condition 1, condition 2
    c1 = num_leaves == 1;
    c2 = num_leaves == 2;
    c3 = num_leaves == 10;
    c4 = num_leaves == 100;
    c5 = num_leaves == 500;
```

```
c6 = num_leaves == 1000;
c7 = num_leaves == 2500;
c8 = num_leaves == 7500;
% If we are at a certain number of leaves, show progress
if c1 || c2 || c3 || c4 || c5 || c6 || c7 || c8
    figure(counter);
    title("")
    title(strcat("Image Using ", num2str(num_leaves), " Leaves"), 'FontSize', text_size);
    hold on;
    % y_offset needed to prevent image from being upside down
    y_offset = n + 1;

    for i = [pure_leaves impure_leaves]
        x_1 = R(i).x(1);
    x_2 = R(i).x(2);
    y_1 = y_offset - R(i).y(1);
    y_2 = y_offset - R(i).y(2);
    fill([x_1, x_2, x_2, x_1], [y_1, y_1, y_2, y_2], R(i).val, 'LineStyle', 'none');
    end
    % Fit image properly and remove axis information
    axis([min(cols) max(cols) min(rows) max(rows)]);
    axis off;
    counter = counter + 1;
end
% Get the oldest leaf since this is FIFO queue with level-order
% traversal
leaf = R(impure_leaves(:,1));
% Store the index of the oldest leaf
current_index = leaf.I;
% Find the dimensions and location of the rectangle for this leaf
min_x = leaf.x(1);
max_x = leaf.x(2);
min_y = leaf.y(1);
max_y = leaf.y(2);
% Find the optimal splitting of the rectangle for this leaf
[j_opt, s_opt] = optimal_split_regression(leaf.vals, leaf.cols, leaf.rows);
% j_opt == 1 means x_axis is the split axis
if j_opt == 1
    % Partition the pixels into those with x-values smaller than or
    % equal to j_opt split index, and those with x-values greater than
    one = (leaf.cols <= s_opt);
    two = (leaf.cols > s_opt);
    c1_vals = leaf.vals(one,:);
    c2_vals = leaf.vals(two,:);
% else, if j_opt == 2, y_axis is the split axis
else
    % Partition the pixels similar to with x-values
    one = (leaf.rows <= s_opt);
    two = (leaf.rows > s_opt);
    c1_vals = leaf.vals(one,:);
    c2_vals = leaf.vals(two,:);
end
% Left child
% No split yet for this so initiailize s and j to be empty
R(leaf_index).s = [];
R(leaf_index).j = [];
% No children yet so initialize left and right to be empty
R(leaf_index).left = [];
R(leaf_index).right = [];
% Placeholder values for rectangle to be updated
R(leaf_index).x = [min_x; max_x];
R(leaf_index).y = [min_y; max_y];
% Index for leaf
R(leaf_index).I = leaf_index;
% Pixels and values of pixels for this leaf
R(leaf_index).rows = leaf.rows(one);
R(leaf_index).cols = leaf.cols(one);
R(leaf_index).vals = leaf.vals(one,:);
% Average RGB value of pixels in this leaf
R(leaf_index).val = sum(R(leaf_index).vals, 1)/size(R(leaf_index).vals, 1);
% Set the x-dimensions or y-dimensions to be the left/lower
% part depending on j_opt
if j_opt == 1
    R(leaf_index).x = [min_x; s_opt];
else
    R(leaf_index).y = [min_y; s_opt];
end
% Classify it as pure or impure leaf
if length(R(leaf_index).rows) == 1
    pure_leaves = [pure_leaves leaf_index];
else
    impure_leaves = [impure_leaves leaf_index];
end
% Set its index
R(current_index).left = leaf_index;
% Update index
leaf_index = leaf_index + 1;
% Right child
% No split yet for this so initiailize s and j to be empty
R(leaf_index).s = [];
R(leaf_index).j = [];
% No children yet so initialize left and right to be empty
R(leaf_index).left = [];
R(leaf_index).right = [];
% Placeholder values for rectangle to be updated
R(leaf_index).x = [min_x; max_x];
R(leaf_index).y = [min_y; max_y];
% Index for leaf
R(leaf_index).I = leaf_index;
% Pixels and values of pixels for this leaf
R(leaf_index).rows = leaf.rows(two);
R(leaf_index).cols = leaf.cols(two);
R(leaf_index).vals = leaf.vals(two,:);
% Average RGB value of pixels in this leaf
R(leaf_index).val = sum(R(leaf_index).vals, 1)/size(R(leaf_index).vals, 1);
% Set the x-dimensions or y-dimensions to be the right/upper
```

```
    % part depending on j_opt
    if j_opt == 1
        R(leaf_index).x = [s_opt; max_x];
    else
        R(leaf_index).y = [s_opt; max_y];
    end
    % Classify it as pure or impure leaf
    if length(R(leaf_index).rows) == 1
        pure_leaves = [pure_leaves leaf_index];
    else
        impure_leaves = [impure_leaves leaf_index];
    end
    % Set its index
    R(current_index).right = leaf_index;
    % Update index
    leaf_index = leaf_index + 1;
    % Set s and j for parent of the two children to be s_opt and j_opt
    R(current_index).s = s_opt;
    R(current_index).j = j_opt;
    % Remove parent from list of impure leaves since it is no longer a leaf
    impure_leaves(:,1)= [];
    % Increment number of splits
    num_splits = num_splits + 1;
end
m = max(cols);
n = max(rows);
% Plot maximal tree image
figure(counter);
title("Image Using Maximal Tree", 'FontSize', text_size);
hold on;
y_offset = n + 1;
for i = pure_leaves
    x_1 = R(i).x(1);
    x_2 = R(i).x(2);
    y_1 = y_offset - R(i).y(1);
    y_2 = y_offset - R(i).y(2);
    fill([x_1, x_2, x_2, x_1], [y_1, y_1, y_2, y_2], R(i).val, 'LineStyle', 'none');
end
axis([min(cols) max(cols) min(rows) max(rows)]);
axis off;
```

Code for optimal_split_regression.m

```
function [j_opt,s_opt] = optimal_split_regression(vals, cols, rows)
% This function takes a list of pixels with RGB values and their locations
% within a rectangle and then determines the best way to split this
% rectangle into two rectangles such that the vectorial mean square error
% of the difference between the average RGB value of each rectangle and
% each pixel within that rectangle is minimized


% Input: vals: list of RGB values for pixels in rectangle
% Input: cols: x-axis locations for pixels in rectangle
% Input: rows: y-axis locations for pixels in rectangle
% Output: j_opt: optimal split dimension. An image only has two dimensions,
% so this is either 1 (x-axis) or 2 (y-axis).
% Output: s_opt: optimal split index. This is the exact point along the
% optimal split dimension where we split the rectangle, it will always be a
% midpoint between two pixels along the optimal split dimension.


% Find all possible x and y values and sort them
% for iteration purposes when trying to find optimal splits
x_values = unique(sort(cols));
y_values = unique(sort(rows));
% Dummy value of inf used to make the min_error
% always update correctly on the first split
min_error = Inf;
% Iterate over all x-values midway between two pixels
[s_opt1, min_error1] = min_split_along_dim(cols, vals, min_error, 1);
[s_opt2, min_error2] = min_split_along_dim(rows, vals, min_error, 2);
% Check to see whether the optimal split is along x or y axis
% Update j_opt and s_opt depending on what the optimal split is
if min_error1 <= min_error2
    j_opt = 1;
    s_opt = s_opt1;
    min_error = min_error1;
else
    j_opt = 2;
    s_opt = s_opt2;
    min_error = min_error2;
end
end
```

Code for min_split_along_dim.m

```
function [s_opt, min_error] = min_split_along_dim(coords, vals, min_error, dim)
% In this function, we're checking splits instead of splitting checks...
% Iterate over all possible splits along one dimension of the data

% Input: coords: the coordinates of the pixels along this dimension
% Input: vals: the RGB values of the pixels along this dimension
% Input: min_error: Usually set to Inf, this is the threshold that an
% optimal split must be less than
% Input: dim: which dimension we are testing splits on, 1 for x or 2 for y
% Output: s_opt: the optimal split index
% Output: min_error: the error from the optimal split

% Initial value of Inf for s_opt is just a dummy value
s_opt = Inf;
% Sort the coords and eliminate duplicates to find all possible split
% indices
unique_sorted_coords = unique(sort(coords));
% Iterate over all midpoints between two different coordinates along this
% dimension
for i = 1:(length(unique_sorted_coords) - 1)
    midpoint = [unique_sorted_coords(i) + unique_sorted_coords(i + 1)]/2;
    % Left (or lower) and right (or upper) partitioning
    indices_1 = [coords <= midpoint];
    indices_2 = [coords > midpoint];
    % Partition the pixels
    vals_1 = vals(indices_1,:);
    vals_2 = vals(indices_2,:);
    % Left average
    c1 = sum(vals_1, 1)/size(vals_1, 1);
    % Right average
    c2 = sum(vals_2, 1)/size(vals_2, 1);
    % Subtract the average RGB value of each rectangle from the RGB values
    % of the pixels in that rectangle
    vals_1_normal = vals_1 - c1;
    vals_2_normal = vals_2 - c2;
    % Left and right error are both mean square errors of the RGB values
    % from the average in their respective rectangle
    left_error = sum(sqrt(sum(vals_1_normal.^2, 2)).^2);
    right_error = sum(sqrt(sum(vals_2_normal.^2, 2)).^2);
    % Total error is sum of left anf right error
    total_error = left_error + right_error;
    % If this split has a lower error than other previously seen splits,
    % then it is the most optimal so far.
    if (total_error < min_error) || (min_error == Inf)
        j_opt = dim;
        s_opt = midpoint;
        min_error = total_error;
    end
end
end
```

Code for toy.m

```
% Scale to make fill work
cols = [25; 80; 60];
rows = [25; 10; 80];
vals = [0.1 0.5 0.2; 0.4 0.1 0.3; 0.9 0.2 0.0];
figure(5);
text_size = 20;
title('The Pixels', 'FontSize', text_size);
hold on;
x_size = 1;
y_size = 1;
for i = 1:3
    x_1 = cols(i) - x_size;
    x_2 = cols(i) + x_size;
    y_1 = rows(i) - y_size;
    y_2 = rows(i) + y_size
    fill([x_1, x_2, x_2, x_1], [y_1, y_1, y_2, y_2], vals(i,:), 'LineStyle', 'none');
end
```

```
%axis([min(cols) max(cols) min(rows) max(rows)]);
% While loop that creates the tree. Will stop running when we either have
% the maximal tree or we have done the maximum number of splits
counter = 1;
impure_leaves = [1];
pure_leaves = [];
m = max(cols);
n = max(rows);
min_x = min(cols);
max_x = max(cols);
min_y = min(rows);
max_y = max(rows);
% Initialize the leaf of the tree
% The split index
R(1).s = [];
% The split dimension
R(1).j = [];
% The left child node index
R(1).left = [];
% The right child node index
R(1).right = [];
% The node index
R(1).I = 1;
% The x dimensions that this node has for the image
R(1).x = [min_x; max_x];
% The y dimensions that this node has for the image
R(1).y = [min_y; max_y];
% The y-axis values for the pixels in this node
R(1).rows = rows;
% The x-axis values for the pixels in this node
R(1).cols = cols;
% The RGB values for the pixels in this node
R(1).vals = vals;
% The average RGB value for the pixels in this node
R(1).val = mean(vals);
leaf_index = 2;
num_splits = 0;
while ~isempty(impure_leaves)
    % Calculate number of leaves to determine whether to show intermediate
    % progress
    num_leaves = length(impure_leaves) + length(pure_leaves);
    % Show image at various stages
    % c1, c2, etc. = condition 1, condition 2
    c1 = num_leaves == 1;
    c2 = num_leaves == 2;
    c3 = num_leaves == 10;
    c4 = num_leaves == 100;
    c5 = num_leaves == 500;
    c6 = num_leaves == 1000;
    c7 = num_leaves == 2500;
    c8 = num_leaves == 7500;
    % If we are at a certain number of leaves, show progress
    if c1 || c2 || c3 || c4 || c5 || c6 || c7 || c8
```

```
        figure(counter);
        title("")
        title(strcat("Image Using ", num2str(num_leaves), " Leaves"), 'FontSize', text_size);
        hold on;
        for i = [pure_leaves impure_leaves]
            x_1 = R(i).x(1);
        x_2 = R(i).x(2);
        y_1 = R(i).y(1);
        y_2 = R(i).y(2);
        fill([x_1, x_2, x_2, x_1], [y_1, y_1, y_2, y_2], R(i).val, 'LineStyle', 'none');
        end
        % Fit image properly and remove axis information
        axis([min(cols) max(cols) min(rows) max(rows)]);
        axis off;
        counter = counter + 1;
    end
    % Get the oldest leaf since this is FIFO queue with level-order
    % traversal
    leaf = R(impure_leaves(:,1));
    % Store the index of the oldest leaf
    current_index = leaf.I;
    % Find the dimensions and location of the rectangle for this leaf
    min_x = leaf.x(1);
    max_x = leaf.x(2);
    min_y = leaf.y(1);
    max_y = leaf.y(2);
    % Find the optimal splitting of the rectangle for this leaf
    [j_opt, s_opt] = optimal_split_regression(leaf.vals, leaf.cols, leaf.rows);
    % j_opt == 1 means x_axis is the split axis
    if j_opt == 1
        % Partition the pixels into those with x-values smaller than or
        % equal to j_opt split index, and those with x-values greater than
        one = (leaf.cols <= s_opt);
        two = (leaf.cols > s_opt);
        c1_vals = leaf.vals(one,:);
        c2_vals = leaf.vals(two,:);
    % else, if j_opt == 2, y_axis is the split axis
    else
        % Partition the pixels similar to with x-values
        one = (leaf.rows <= s_opt);
        two = (leaf.rows > s_opt);
        c1_vals = leaf.vals(one,:);
        c2_vals = leaf.vals(two,:);
    end
    % Left child
    % No split yet for this so initiailize s and j to be empty
    R(leaf_index).s = [];
    R(leaf_index).j = [];
    % No children yet so initialize left and right to be empty
    R(leaf_index).left = [];
    R(leaf_index).right = [];
    % Placeholder values for rectangle to be updated
    R(leaf_index).x = [min_x; max_x];
```

```
R(leaf_index).y = [min_y; max_y];
% Index for leaf
R(leaf_index).I = leaf_index;
% Pixels and values of pixels for this leaf
R(leaf_index).rows = leaf.rows(one);
R(leaf_index).cols = leaf.cols(one);
R(leaf_index).vals = leaf.vals(one,:);
% Average RGB value of pixels in this leaf
R(leaf_index).val = sum(R(leaf_index).vals, 1)/size(R(leaf_index).vals, 1);
% Set the x-dimensions or y-dimensions to be the left/lower
% part depending on j_opt
if j_opt == 1
    R(leaf_index).x = [min_x; s_opt];
else
    R(leaf_index).y = [min_y; s_opt];
end
% Classify it as pure or impure leaf
if length(R(leaf_index).rows) == 1
    pure_leaves = [pure_leaves leaf_index];
else
    impure_leaves = [impure_leaves leaf_index];
end
% Set its index
R(current_index).left = leaf_index;
% Update index
leaf_index = leaf_index + 1;
% Right child
% No split yet for this so initiailize s and j to be empty
R(leaf_index).s = [];
R(leaf_index).j = [];
% No children yet so initialize left and right to be empty
R(leaf_index).left = [];
R(leaf_index).right = [];
% Placeholder values for rectangle to be updated
R(leaf_index).x = [min_x; max_x];
R(leaf_index).y = [min_y; max_y];
% Index for leaf
R(leaf_index).I = leaf_index;
% Pixels and values of pixels for this leaf
R(leaf_index).rows = leaf.rows(two);
R(leaf_index).cols = leaf.cols(two);
R(leaf_index).vals = leaf.vals(two,:);
% Average RGB value of pixels in this leaf
R(leaf_index).val = sum(R(leaf_index).vals, 1)/size(R(leaf_index).vals, 1);
% Set the x-dimensions or y-dimensions to be the right/upper
% part depending on j_opt
if j_opt == 1
    R(leaf_index).x = [s_opt; max_x];
else
    R(leaf_index).y = [s_opt; max_y];
end
% Classify it as pure or impure leaf
if length(R(leaf_index).rows) == 1
```

```
            pure_leaves = [pure_leaves leaf_index];
        else
            impure_leaves = [impure_leaves leaf_index];
        end
        % Set its index
        R(current_index).right = leaf_index;
        % Update index
        leaf_index = leaf_index + 1;
        % Set s and j for parent of the two children to be s_opt and j_opt
        R(current_index).s = s_opt;
        R(current_index).j = j_opt;
        % Remove parent from list of impure leaves since it is no longer a leaf
        impure_leaves(:,1)= [];
        % Increment number of splits
        num_splits = num_splits + 1;
    end

    figure(counter);
    title("")
    title(strcat("Image Using 3 Leaves: Maximal Tree"), 'FontSize', text_size);
    hold on;
    for i = pure_leaves
        x_1 = R(i).x(1);
    x_2 = R(i).x(2);
    y_1 = R(i).y(1);
    y_2 = R(i).y(2);
    fill([x_1, x_2, x_2, x_1], [y_1, y_1, y_2, y_2], R(i).val, 'LineStyle', 'none');
    end
    % Fit image properly and remove axis information
    axis([min(cols) max(cols) min(rows) max(rows)]);
    axis off;
    counter = counter + 1;
```