# Final Report

Zachary LeClaire

September 18, 2022

## Introduction

From large businesses to your family's photo albumn, data corruption and loss are issues that can impact nearly everyone, even those that don't use computers. Following from the later example, image data loss can result from many different issues, including software bugs, issues with computer hardware, and environmental factors like cosmic rays. A useful tool to recover lost data are regression trees, which recursively partition the image into branches, forming a tree structure. In this report, the development of a regression tree tool will be discussed along with its application to a sparse image, "MysteryImage.mat", to analyze the technique's effectiveness.

## Part 1: Optimal Splitting

The first challenge in implementing a regression tree is finding the best split in the current branch of the tree. For the first iteration, this would mean the best split for the image entire image that partitions it into two parts: left and right branches. For this problem, a way of determining the best split is to find the split that minimizes the mean-squared error of the pixels with the mean color in that branch. As as a result, the problem can computationally performed by iterating through each possible split and finding the split with the minimum minimum mean-squared error associated with it. We can evaluate the cost of a horizontal and vertical split with the expressions:

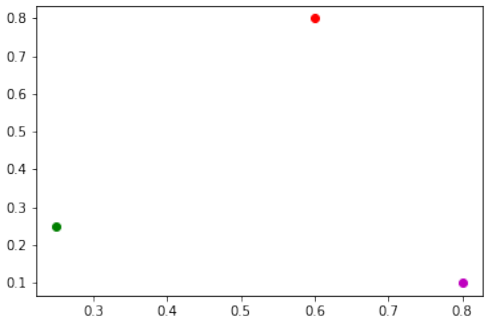
$$F_x(s, c1, c2) = \sum_{x^{(j)} <= s} \|v^{(j)} - c_1\|^2 + \sum_{x^{(j)} > s} \|v^{(j)} - c_2\|^2$$

$$F_y(s, c1, c2) = \sum_{y^{(j)} <= s} \|v^{(j)} - c_1\|^2 + \sum_{y^{(j)} > s} \|v^{(j)} - c_2\|^2$$

These cost expressions assume that $c_1$ and $c_2$ are the mean colors of the left and right branches, respectively. Additionally, $v^{(j)}$ refers to the color of the jth pixel in the branch.

To determine how this expression would be calculated, consider an example with the three following points:

| X | Y | color |
|---|---|---|
| 0.25 | 0.25 | [0.1,0.5,0.2] |
| 0.8 | 0.1 | [0.4,0.1,0.3] |
| 0.6 | 0.8 | [0.9,0.2,0.0] |




Seeing there there are 3 unique coordinates, there are 4 possible split values. We can find each split by taking the mean of two adjacent coordinates. In other words, if the list of x-coordinates were to be sorted, each adjacent pair of coordinates would have an associated split value between these points. Resulting from this, we can find the following table:

For the MSE calculations of the first split, we first need to find the means of the left and right branches. If n,m are the number of pixels in the branch, then the means for the vertical splits can be found with the expressions:

$$c_1 = 1/n * \sum_{x^{(j)} <= s} v^{(j)}$$

$$c_2 = 1/m * \sum_{x^{(j)} > s} v^{(j)}$$

The means for horizontal splits are similar, with the y-coordinates being compared to the split instead of the x-coordinates. The means for this toy example were calculated using this formula and listed in the table below. The detailed calculations are listed in the appendix above the code for this project.

| Number | Direction | Split | $c_1$ | $c_2$ | Mean-Squared-Error (MSE) |
|---|---|---|---|---|---|
| 1 | Horizontal | 0.175 | [0.4,0.1,0.3] | [0.5,0.35,0.1] | 0.385 |
| 2 | Horizontal | 0.525 | [0.25,0.3,0.25] | [0.9,0.2,0.0] | 0.13 |
| 3 | Vertical | 0.425 | [0.1,0.5,0.2] | [0.65,0.15,0.15] | 0.175 |
| 4 | Vertical | 0.7 | [0.5,0.35,0.1] | [0.4,0.1,0.3] | 0.385 |

Examining the results in this table shows that the best split is a horizontal split at 0.525 because it has the smallest the MSE of all possible splits.

In terms of the actual implementation of optimal splitting, we can divide the process into the following steps:

- Find all possible vertical and horizontal splits
- Group points into left and right branches
- Calculate means of each branch
- Calculate MSE's of each split, choosing the smallest option

Firstly, the possible vertical and horizontal splits can be found efficiently by sorting the list of x and y coordinates separately, storing the values shifted to the left by 1 index, cutting off the last element of each array, and taking their means. The Python implementation of this procedure is:

```
x_sorted = np.sort(x_data)
```

```
y_sorted = np.sort(y_data)

#get split values by summing array with shifted array by 1
and taking mean of every entry
x_splits = 0.5 * (x_sorted[:-1]+np.roll(x_sorted,-1)[:-1])
y_splits = 0.5 * (y_sorted[:-1]+np.roll(y_sorted,-1)[:-1])
```

Because Numpy's sort algorithm uses Quicksort, this should in general be an average case run time complexity of $O(n * log_2(n))$ for n number of elements in the array. Additionally, approaching this problem with vectorization is more efficient than using for loop, as computer hardware is better at handling bulk operations where multiple variables are pulled into CPU cache than single operations.

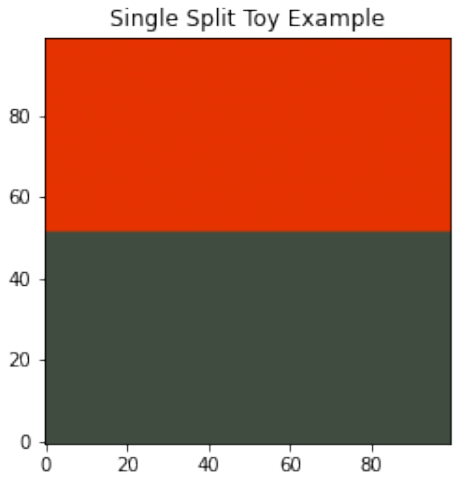
Once the possible splits have been found, the MSE of each split is found by first grouping points base on which side of split they exist on. For the horizontal split cost function, the code to find the groups of left and right values is below:

```
left_values = vals[np.where(y_values <= s)]
right_values = vals[np.where(y_values > s)]
```

After the values are grouped, we can find the difference between the mean of the region and each pixel. Then, the MSE can be computed in only one line. The implementation for this is:

```
c1 = np.mean(left_values, axis = 0)
c2 = np.mean(right_values, axis = 0)
left_values -= np.tile(c1, (len(left_values), 1))
right_values -= np.tile(c2, (len(right_values), 1))
cost = np.sum(np.square(np.linalg.norm(left_values, axis=1))) +
np.sum(np.square(np.linalg.norm(right_values, axis=1)))
```

The result of the first split from this Python Script are shown below:




The split appears to match the results from our hand calculations, with a horizontal split at 0.525. It should be noted that a 100x100 pixel canvas was used for this visualization, so each 0.01x0.01 rectangle in the 1.0 by 1.0 grid is mapped to one pixel. Additionally, the colors match the calculated means of the branches. To properly render the image, the leaf nodes, or nodes that have no children, should

be filled with the color of the mean colors of the pixels inside. In this example, only one split occurs, creating a pure leaf of red and a mixed leaf with both violet and green.

# Part 2: Regression

In order to perform regression on an image, branches are split optimally until either a threshold is reached (eg. maximum of 100 iterations), or every branch is pure. It is often better to follow the former case, as maximal trees with only pure branches are more prone to outliers and noise. In the case of limiting number of iterations, this kind of splitting would be implemented using a non-recursive method that keeps track of current leaf nodes. In this project, each new mixed leaf node was added to a First-In First-Out (FIFO) queue, providing relatively even priority to each branch. An alternative approach would be to use a priority queue weighted by MSE, but the results of using a standard FIFO queue were satisfactory so this approach was not pursued. The main benefit this would likely provide is possibly better results for a smaller number of iterations, as messier branches with higher MSE's would be prioritized for splitting.

**Algorithm 1** Regression Tree

```
Parameters:
arrays X,Y,V, queue q, constant MAXITTER, leaf list
while i < MAX_ITTER and q.size > 0 do
    pop node off queue
    find optimal split in new node
    create left and right nodes
    if node is impure then
        push node to queue
    else
        add node to leaf node list
    end if
    i = i+1
end while
while q.size > 0 do
    pop node off queue
    add node to leaf list
end while
for node in leaf list do
    paint canvas color of mean of pixels for node
end for
```

To encapsulate multiple pieces of information together, including the border pixels of the node and the pixels inside, a module holding a Node class was created. This made it easy to queue a single node instead of dealing with parallel queues that stored primitive data types. This class is stored in the Tree Module (Tree.py), which can be found in the appendix.

Once the arrays of coordinates and color values are initialized, the script initializes the queue before entering the while loop. First, it created the first node, which is simply the entire branch. Then, the node is added to the queue.

```
init_node = Node(X, Y, V, 0, x_dim, 0, y_dim)
q.append(init_node)
```

Once the loop as concluded running until either the maximum number of iterations or until it has reached a maximal tree, then every element currently in the queue is added to the leaf list, which holds all pure leaves until the loop finishing running. Then, each node in the leaf list is used to paint with

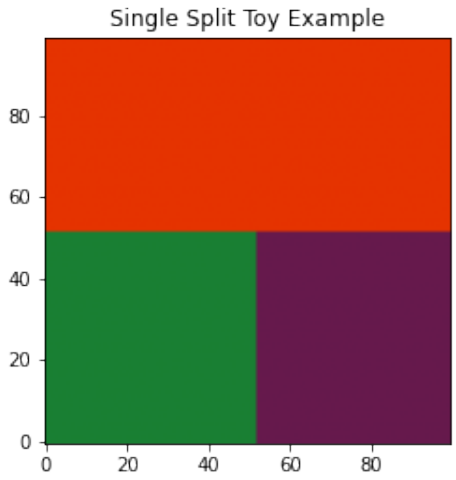
its mean color. Once the image matrix stores each pixel value from the nodes, it is then visualized using matplotlib's imshow method.

```
img = np.zeros((x_dim, y_dim, 3))
for node in leaf_list:
    img[node.x_min: node.x_max,  node.y_min:node.y_max] = np.mean(node.vals, axis = 0)
while len(q) >0:
    node = q.pop(0)
    img[node.x_min: node.x_max,  node.y_min:node.y_max] = np.mean(node.vals, axis = 0)
return img
```

Considering the toy example, the second iteration will only consider the lower green-ish grey branch because it is not pure ($MSE > 0$). The possible splits are listed below:

| Number | Direction | Split | $c_1$ | $c_2$ | Mean-Squared-Error (MSE) |
|---|---|---|---|---|---|
| 1 | Horizontal | 0.175 | [0.4,0.1,0.3] | [0.1,0.5,0.2] | 0 |
| 2 | Vertical | 0.525 | [0.1,0.5,0.2] | [0.4,0.1,0.3] | 0 |

Examining these results show that either split is valid, as they both have MSE's of 0 because they're pure. In the event of equivalently good splits, the choice doesn't matter. In this implementation of regression trees, the first split considered will be the one choice by the regression function. Plotting this second second iteration, we get following result:




## Part 3: Mystery Image

Now, applying this technique to do regression on the mystery image, we first need to process the image into a form that the regression script can handle. This is relatively straightforward with the scipy library's loadmat function.

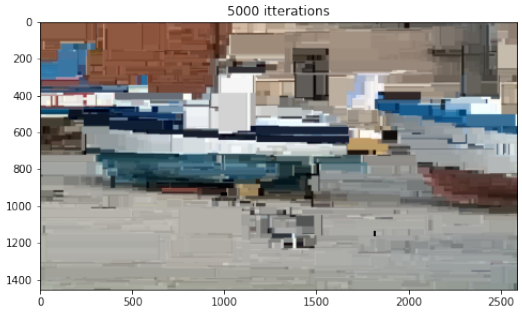
```
image = scipy.io.loadmat('MysteryImage.mat')

#extract compressed matrix from dictionaries
```

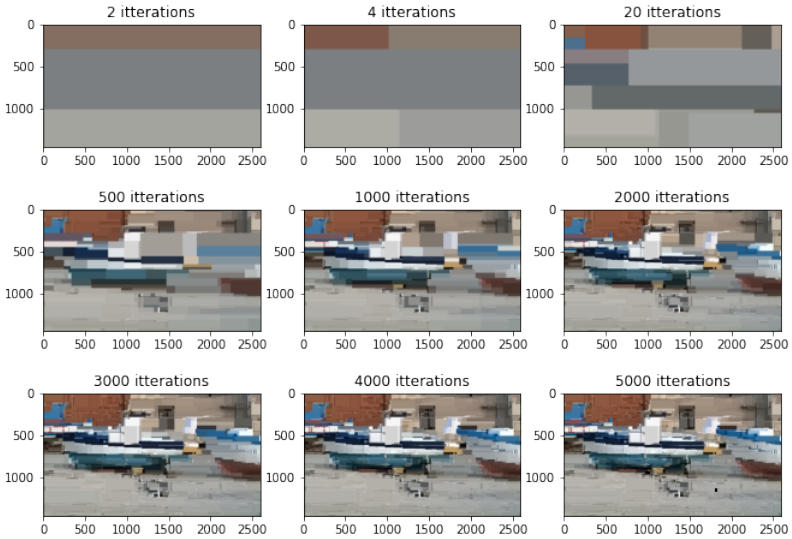
```
#columns: y-axis; rows: x-axis; vals: [r,g,b]
Y = image['cols'][...,0]
X = image['rows'][...,0]
V = image['vals']
```

Note that the "[...,0]" is necessary to pull the x and y coordinates out of a nested array. Otherwise, the arithmetic operations are all performed on singleton arrays, which causes errors for certain operations in Python and results in incorrect results. If using any particular file type with this code, it is important to ensure that X and Y are single dimensional arrays and V is a 1d array of 1x3 arrays storing color values.

Running this up until 5000 iterations yields the following image:




This image seems to resemble a boat. Although missing many detaills, the image has recognizable features, including the brick wall in the background and the two boats featured in this image. Plotting the results at various stages in the regression process shows the boat's shape slowly take form:



After approximately 100-500 iterations the boat becomes recognizable. Each subsequent iteration adds details which would be otherwise missed. For example, the door behind the the boat is only recognizable after 2000 iterations. This result seems to be effective for recovering information from images that have lost a large number of pixels. It is able to construct recognizable features, but as a result it's somewhat sensitive to noise. If working with nosier data, it would seem best to use a smaller number of iterations, as the error in the pixels would become more significantly weighted in smaller nodes.

# Appendix

Calculations for MSE's of first split: $MSE_1 = 0 + (0.5-0.1)^2 + (0.35-0.5)^2 + (0.1-0.2)^2 + (0.5-0.9)^2 + (0.35-0.2)^2 + (0.1-0)^2 = 0.385$

$MSE_2 = (0.25-0.1)^2 + (0.3-0.5)^2 + (0.25-0.2)^2 + (0.25-0.4)^2 + (0.3-0.1)^2 + (0.25-0.3)^2 + 0 = 0.13$

$MSE_3 = 0 + (0.65-0.4)^2 + (0.15-0.1)^2 + (0.15-0.3)^2 + (0.65-0.9)^2 + (0.15-0.2)^2 + (0.15-0.0)^2 = 0.175$

$MSE_4 = (0.5-0.1)^2 + (0.35-0.5)^2 + (0.1-0.2)^2 + (0.5-0.9)^2 + (0.35-0.2)^2 + (0.1-0)^2 + 0 = 0.385$

Calculations for MSE's of second split:

$MSE_1 = (0.4-0.4)^2 + (0.1-0.1)^2 + (0.3-0.3)^2 + (0.1-0.1)^2 + (0.5-0.5)^2 + (0.2-0.2)^2 = 0$
$MSE_2 = (0.4-0.4)^2 + (0.1-0.1)^2 + (0.3-0.3)^2 + (0.1-0.1)^2 + (0.5-0.5)^2 + (0.2-0.2)^2 = 0$

Tree Module:

```
    '''
Module for holding Binary Tree
Desc: Binary Tree we use for storing results of tree regression
'''

import numpy as np

class Node:

    #constructor of Node class
    def __init__(self, x_elements, y_elements, values, x_min, x_max, y_min, y_max):
        self.x_elements = x_elements
        self.y_elements = y_elements
        self.vals = values
        self.x_min = x_min
        self.x_max = x_max
        self.y_min = y_min
        self.y_max = y_max

    #useful information about the node
    def print_me(self):
        print(self.x_elements)
        print(self.y_elements)
        print(f"x:({self.x_min}, {self.x_max}); y:({self.y_min}, {self.y_max})")
        print(self.vals)
```

Regression Tree Python Script:

```
# %%
```

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
from matplotlib import colors
import threading, queue
import scipy.io


from Tree import Node


# %%
#split s means we check all y-values to see if they're above or below the line
# %%
#split s means we check all y-values to see if they're above or below the line
'''
split cost functions
Desc: finds cost of a split
parameters:
s: split value
c1: mean color left/below split [r,g,b]
c2: mean color right/above split [r,g,b]
'''
def hor_split_cost(s, y_values, vals):
    #get values left

    left_values = vals[np.where(y_values <= s)]
    right_values = vals[np.where(y_values > s)]

    c1 = np.mean(left_values, axis = 0)
    c2 = np.mean(right_values, axis = 0)

    left_values -= np.tile(c1, (np.shape(left_values)[0], 1))
    left_cost = np.sum(np.square(np.linalg.norm(left_values, axis=1)))

    right_values -= np.tile(c2, (np.shape(right_values)[0], 1))
    right_cost = np.sum(np.square(np.linalg.norm(right_values, axis=1)))
    cost = right_cost + left_cost
    return cost, c1, c2, right_cost, left_cost

def vert_split_cost(s, x_values, vals):
    #get values left
    left_values = vals[np.where(x_values <= s)]
    right_values = vals[np.where(x_values > s)]
    c1 = np.mean(left_values, axis = 0)
    c2 = np.mean(right_values, axis = 0)
    #print(len(left_values))
    #print(c1)
    left_values -= np.tile(c1, (np.shape(left_values)[0], 1))
    left_cost = np.sum(np.square(np.linalg.norm(left_values, axis=1)))

    right_values -= np.tile(c2, (np.shape(right_values)[0], 1))
    right_cost = np.sum(np.square(np.linalg.norm(right_values, axis=1)))
```

```
    #print(left_values)
    cost = left_cost + right_cost
    return cost, c1, c2, right_cost, left_cost

# %%
'''
Returns direction & optimal split value
'''
def OptimalSplitRegression(x_data,y_data,vals):

    #pre-sort X & Y values:
    x_sorted = np.sort(x_data)
    y_sorted = np.sort(y_data)

    #get split values by summing array with shifted array by 1 to right and taking mean of every entr
    x_splits = 0.5 * (x_sorted[:-1]+np.roll(x_sorted,-1)[:-1])
    y_splits = 0.5 * (y_sorted[:-1]+np.roll(y_sorted,-1)[:-1])

    min_cost = float('inf')
    min_c1: np.array
    min_c2: np.array
    min_direction = False
    min_s = 0
    min_LC = 0
    min_RC = 0
    #print("starting splits")
    #find min cost split among both x and y splits
    for x_split in x_splits:
        cost, c1, c2, left_cost, right_cost = vert_split_cost(x_split, x_data, vals)
        #print(f'x_split: {x_split}; cost: {cost}')
        if(cost <= min_cost):
            min_s = x_split
            min_cost = cost
            min_c1 = c1
            min_c2 = c2
            min_direction = True
            min_LC = left_cost
            min_RC = right_cost

    #print("Done with X splits")
    for y_split in y_splits:
        cost, c1, c2, left_cost, right_cost = hor_split_cost(y_split, y_data, vals)
        #print(f'y_split: {y_split}; cost: {cost}')
        if(cost <= min_cost):
            min_s = y_split
            min_cost = cost
            min_c1 = c1
            min_c2 = c2
            min_direction = False
            min_LC = left_cost
            min_RC = right_cost
```

```
    return min_s, min_cost, min_c1, min_c2, min_direction, min_LC, min_RC

# %%
#get matlab data as nested dictionary
image = scipy.io.loadmat('MysteryImage.mat')

#extract compressed matrix from dictionaries

#columns: y-axis; rows: x-axis; vals: [r,g,b]
Y = image['cols'][...,0]
X = image['rows'][...,0]
V = image['vals']

# number of pixels
n = np.shape(X)[0]

# %%
#Code for Regression Tree:
def regression(MAX_ITTER):
    x_dim = 1456
    y_dim = 2592

    #stores leaves to consider
    q = []
    # X_points, Y_points, x_min, x_max, y_min, y_max, color
    init_node = Node(X, Y, V, 0, x_dim, 0, y_dim)

    q.append(init_node)
    itter = 0
    img = np.zeros((x_dim, y_dim, 3))

    #q.qsize() function is inaccurate, so we use our own count
    leaf_list = []
    while itter < MAX_ITTER and len(q) > 0:
        itter += 1
        curr_node = q.pop(0)
        s, cost, c1, c2, direction, min_LC, min_RC = OptimalSplitRegression(curr_node.x_elements,curr
        #print(curr_node.x_elements, curr_node.y_elements)
        #write to our image with the current split
        #print(f"s: {s}, c1: {c1}, c2: {c2}")
        left_node = None
        right_node = None
        if(not direction):
            #horizontal cut:
            y_left = curr_node.y_elements[curr_node.y_elements <= s]
            y_right = curr_node.y_elements[curr_node.y_elements > s]

            lefts = np.where(curr_node.y_elements <= s)
            rights = np.where(curr_node.y_elements > s)

            left_values = curr_node.vals[lefts]
            right_values = curr_node.vals[rights]
```

```
            #create children node we push to the queue:
            left_node = Node(curr_node.x_elements[lefts], y_left, left_values, curr_node.x_min, curr_
            right_node = Node(curr_node.x_elements[rights], y_right, right_values, curr_node.x_min, c
            #print("len lengths: ", lefts, rights, np.size(lefts))
        else:
            #vertical cut:
            x_left = curr_node.x_elements[curr_node.x_elements <= s]
            x_right = curr_node.x_elements[curr_node.x_elements > s]
            lefts = np.where(curr_node.x_elements <= s)
            rights = np.where(curr_node.x_elements > s)
            left_values = curr_node.vals[lefts]
            right_values = curr_node.vals[rights]

            #create children node we push to the queue:
            left_node = Node(x_left, curr_node.y_elements[lefts], left_values, curr_node.x_min, int(s
            right_node = Node(x_right, curr_node.y_elements[rights], right_values, int(s), curr_node.
            #print("len lengths: ", lefts, rights, np.size(lefts))
        #Enqueue nodes or if pure add them to list of lead nodes
        #left_node.print_me()
        #right_node.print_me()
        if(np.size(lefts) > 1 and min_LC > 0):
            q.append(left_node)
        else:
            leaf_list.append(left_node)
        if(np.size(rights) > 1 and min_RC > 0):
            q.append(right_node)
        else:
            leaf_list.append(right_node)
        #END WHILE LOOP
    print(f"itteration: {itter}")
    for node in leaf_list:
        img[node.x_min: node.x_max,  node.y_min:node.y_max] = np.mean(node.vals, axis = 0)
    while len(q) >0:
        #print(q.qsize())
        node = q.pop(0)
        #print(node.x_min, node.x_max,  node.y_min,node.y_max, node.vals)
        img[node.x_min: node.x_max,  node.y_min:node.y_max] = np.mean(node.vals, axis = 0)
    return img

# %%
images = []
list = [2, 4, 20, 500, 1000, 2000, 3000, 4000, 5000]
for i in list:
    images.append(regression(i))

# %%
plt.figure(figsize=(11, 8))
for i in range(0, 9):
    ax = plt.subplot(3,3,i+1)
    title = f"{list[i]} itterations"
    ax.title.set_text(title)
    plt.imshow(images[i], origin = "upper")
plt.savefig("regression_image-3x3.png")
```

```
#plt.imshow(images[0])
```