Math 444: Project 7

Levent Batakci

May 3, 2021

In this project, we leveraged tree regression to reconstruct an image that has been compressed by retaining a tiny percentage of its pixels. All of the mentioned implementations can be found on the public github repo.

Compressed Image and Tree Regression Background

In this project, we reconstruct a compressed image. Specifically, a 2736×3648 pixel image has been compressed to retain only 15000 of its pixels. Computing the ratio, we see that this is roughly $100 \times \frac{15000}{2736 \times 3648} = 100 \times \frac{15000}{9980928} \approx 0.15\%$ of the original pixels!

The image is an rgb color image, so each of the retained pixels contains a triplet representing its color value. Essentially, what we will be doing is partitioning the image into rectangular regions and coloring the pixels in a given region with the average value of the known (retained) pixels in that region.

To accomplish this partitioning, we will use tree regression. Tree regression is a comparison-based way to classify data points based on their attributes.

Moreover, we will be computing and using the maximal tree. A maximal tree is one in which none of the leaves (rectangles with no subrectangles in the tree) have an associated misclassification error. In the context of this project, this means that all of the retained pixels in a given leaf will have the same color. Note that a maximal tree always exists under the assumption that no two data points with identical attributes have different classes.

As for attributes, we will simply be identifying pixels by their row and column indices. Note that this pair essentially creates a coordinate point. With these two values used as attributes, we see that no two retained pixels will have the same attributes and so a maximal tree is guaranteed.

We will provide a high-level description of how the maximal tree is computed. Initially, the image is taken as is, and there is considered to only be one rectangle - the entire image. We keep track of leaves (rectangles without subrectangles in the tree) which have non-zero misclassification error. This leaves are said to be impure.

While we still have impure leaves, we pick one and divide it into two subrectangles in a way to minimize the sum of the errors in the resulting rectangles (which are clearly leaves). One thing to note is that the split will either be horizontal ($k = 1$) or vertical ($k = 2$). Moreover, since the number of pixels in a rectangle is finite, we only need to consider a finite number of splits. Indeed, we only consider splits that occur halfway between two known pixels. Specifically, there are no more than $2(x - 1)$ splits to consider in a rectangle which contains x known pixels. Denoting c_1 and c_2 to be the average rgb values in the left and right subrectangles induced by a split at s in direction k, we associate an error

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

where t represents the data points attributes and x represents the rgb values.

So we keep splitting impure leaves and updating the list until none remain. At this point, we are left with a maximal tree. From there, we can reconstruct image simply by coloring the resulting rectangles with their associated rgb values.

One thing to note is that the maximal tree very well may be space and time inefficient. Our code has been optimized to only store the leaves of the tree and reconstruct the image speedily. Specifically, we designed a leaf data structure which utilizes MATLAB's ability to quickly perform element-wise matrix operations. Essentially, the image is reconstructed in chunks at a time. So our approach is very time efficient, and fairly space efficient too.

A more general solution is to prune the tree. That is, remove leaves/nodes to balance the complexity of the tree with the quality of its results. We implemented a pruning algorithm but were unable to verify it's results. Specifically, we ran into some issues when it came to runtime and deciding how to quantify misclassification. Moreover, we are overwhelmed with other projects and decided that optimizing the maximal tree would be a sufficient solution for now.

Results

Before we show the reconstruction, we want to show the given image to really give the reader an understanding of how minuscule a percentage 0.15% is. Below, **Figure 1** shows the compressed image. Note that 0.15% retention means that only 15 out of every 10000 pixels are known. It's likely that you will have to zoom in to see the given pixels - ones not given are painted black.

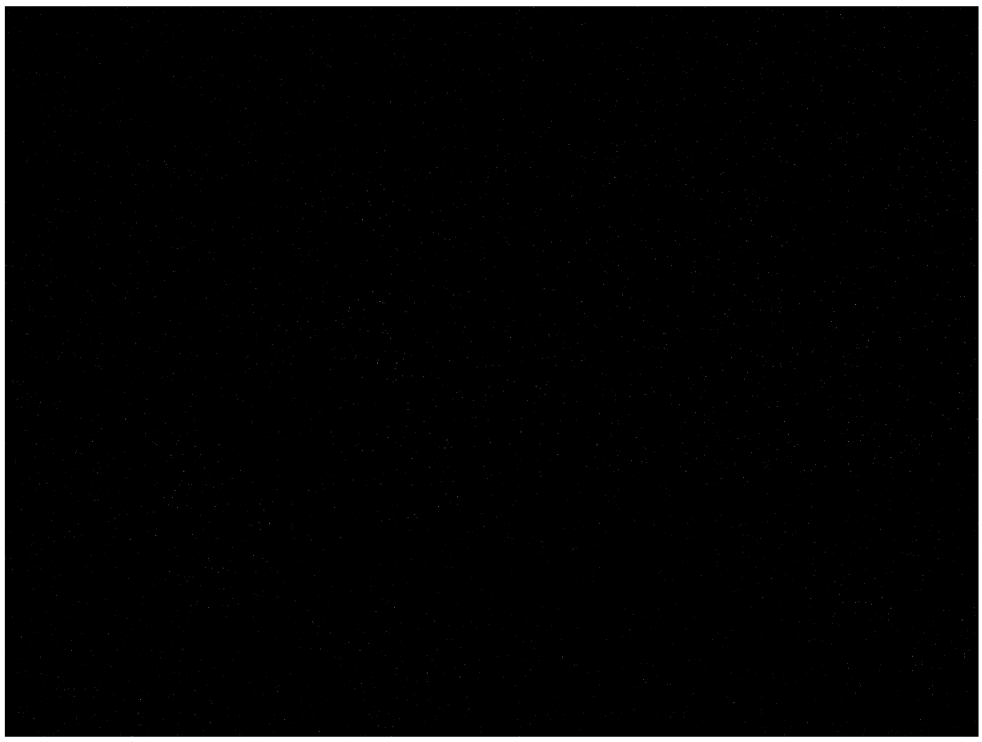

Figure 1: Compressed Image

Personally, we can decipher next to nothing from this image. The pixels are visible, but we have no idea what the image is even of. As discussed, we used tree regression to reconstruct this seemingly meaningless compression. On the next page, **Figure 2** shows our results.

Figure 2: Reconstructed Image

Our first reaction to this reconstruction was amazement. While the amount of detail is subpar as compared to most modern photographs, we found it to be simply incredible that this much information could be recovered. Clearly, the image is of a lighthouse by a house. Even the sky has a gradient of blue.

Overall, we consider this reconstruction to be a huge success. It's worth noting that even if we had managed to pruned the tree, the results would likely have just been worse. Indeed, pruning is a procedure undertaken for time/space efficiency - not for result quality. In the future, we hope to explore ways to smooth the resulting image so as to create a more natural (and less blocky) looking result.

Appendix

As always, feel free to access the code at the public github repo. The Node data class and associated algorithms are not included because there are many helper functions and it would not be very readable on an LaTeX document.

1 Reconstruction Driver

```
%Levent Batakci
%Assignment 7, MATH444
%Tree Regression Image recovery

%Load the image
```

```
%load Nodes
load MysteryImage.mat
load MaximalTree.mat %PRECOMPUTED AND STORED, BY ME
disp("Loaded");

%Compute the maximal tree
%[root, Nodes, Leaves] = MaximalTree(rows, cols, vals, m, n);

%Graph the given
fullImage =  zeros(m,n,3);

for i = 1:15000
    fullImage(rows(i), cols(i),:) = vals(i);
end
imshow(fullImage);

keyboard;
clf
fullImage =  zeros(m,n,3);

%Graph the reconstruction
for i = 1:size(Leaves,2)
    leaf = Leaves{i};
    r = rowRange(leaf);
    c = colRange(leaf);

    color = zeros(1,1,3);
    color(1,1,:) = leaf.c;
    fullImage(r,c,:) = fullImage(r,c,:) + color;

    if(mod(i,2000) == 0)
        imshow(fullImage);
    end
end
imshow(fullImage);

disp("DONE!")
```

2 Maximal Tree Code

```
function [root, Nodes, Leaves] = MaximalTree(rows, cols, vals, m, n)
    root = Node(rows, cols, vals, m, 1, n, 1);
    Nodes = cell(1);
    Nodes{1} = root;

    %GROW THE TREE !
    Lpure = [];
    Lmixed = [1];
    count=1;

    Leaves = cell(1);
    i = 1;
```

```
    while numel(Lmixed) ~= 0
        index = Lmixed(1);
        chosen = Nodes{index}; %Choose mixed node

        %Adjust Lmixed
        if(numel(Lmixed) == 1)
            Lmixed = [];
        else
            Lmixed = Lmixed(2:end);
        end

        %Optimally split the node
        chosen = splitNode(chosen);

        %Check the children
        L = chosen.left;
        count=count+1;
        if(getErr(L) ~= 0)
            Lmixed(end+1) = count;
        else
            Lpure(end+1) = count;
            Leaves{i} = L;
            i=i+1;
        end
        Nodes{count} = L;

        R = chosen.right;
        count=count+1;
        if(getErr(R) ~= 0)
            Lmixed(end+1) = count;
        else
            Lpure(end+1) = count;
            Leaves{i} = R;
            i=i+1;
        end
        Nodes{count} = R;

        if(mod(count+1,100) == 0)
            disp(count)
        end
    end

    root = Nodes{1};
end
```