Homework 4

By: Daniel Speiser and Gideon Glass

December 1^{st}, 2015

Contents

1 Part 1: Written Exercises

1.1 Association Rule Learning

Association Rule Learning	
Item Set	Number of purchases containing at least these items
{}	927,125
Dog Food	80,915
Cat Food	185,279
Cat Litter	130,122
Cat Food, Dog Food	60,159
Cat Food, Cat Litter	120,091
Burgers	29,751
Burgers, Vitamin C Tablets	231
Vitamin C Tablets, Artisian Tap Water	25
Burgers, Buns, Ketchup	15,293
⋮	⋮

A. Q: Sanity check: How many purchases did Nile.com fulfill last month?
A: 927,125

B. Q: Sanity check: How many customers purchased Vitamin C tablets in addition to their other items?
Lower bound = 231
Upper bound = 927,125 - (29,751 - 231) = 897,605
Calculated by purchases which definitely included Vitamin C Tablets minus those who definitely did NOT.

C. Sanity check: How many customers purchased only cat food and nothing else?
Lower bound = 0
Upper bound = (185,279 - 120,091) = 65,188
Calculated by assuming worst case that all {Cat Food, Dog Food} purchases were included in {Cat Food, Cat Litter} transactions.

D. What is the (possible) value of SUPP(Burgers, Vitamin C Tablets, Ketchup)?
Lower bound = 0 (no evidence that transaction necessarily occurs)
Upper bound = $\frac{231}{927,125}$
The number of purchases of {Burgers, Vitamin C, Ketchup} is at most 231 because {Burgers, Vitamin C} is 231.

E. What is the (possible) value of SUPP(Burgers, Buns)? How do you know?
Lower bound = $\frac{15,293}{927,125}$
Based on {Burgers, Buns, Ketchup} we can deduce that at least 15,293 purchases included burgers and buns.
Upper bound = $\frac{29,751}{927,125}$ Based on {Burgers} we can deduce that at most 29,751 purchases included burgers and buns by assuming every burger purchase included a bun.

F. What is the (possible) value of CONF(Burgers $\Longrightarrow$ Vitamin C Tablets)? Does this seem like an interesting promotion opportunity?

$$CONF(Burgers \Longrightarrow VitaminCTablets) = \frac{SUPP(Burgers, VitaminCTablets)}{SUPP(Burgers)}$$

$$= \frac{231}{927,125} \div \frac{29,751}{927,125} = 0.0078$$

With such a low confidence, there doesn't appear to be any interesting promotional opportunities between the two items.

G. What is the (possible) value of CONF(Dog Food, Cat Food $\Longrightarrow$ Cat Litter)?

$$CONF(DogFood, CatFood \Longrightarrow CatLitter) = \frac{SUPP(DogFood, CatFood, CatLitter)}{SUPP(DogFood, CatFood)}$$

Lower bound = 0 (no evidence that transaction {Dog Food, Cat Food, Cat Litter} necessarily occurs)
Upper bound = 1
Based on {Cat Food, Dog Food}, we can deduce that at most 60,159 purchases included {Cat Food, Dog Food, Cat Litter}, thus:

$$= \frac{60,159}{927,125} \div \frac{60,159}{927,125} = 1$$

H. What is the (possible) value of LIFT(Dog Food $\Longrightarrow$ Cat Food)?

$$SUPP(DogFood, CatFood) = \frac{60,159}{927,125}$$

$$SUPP(DogFood) = \frac{80,915}{927,125}$$

$$SUPP(CatFood) = \frac{185,279}{927,125}$$

$$LIFT(DogFood \Longrightarrow CatFood) = \frac{SUPP(DogFood, CatFood)}{SUPP(DogFood) * SUPP(CatFood)}$$

$$= 3.72$$

I. Suppose you wish to run the APriori algorithm with a minimum support of 0.1 (10% of purchases). You begin by gathering the support for all item sets of length 1. Which pairs of items could APriori definitely eliminate using the downward closure property?

$$SUPP(DogFood) = 0.087 < 0.1$$

$$SUPP(Burgers) = 0.032 < 0.1$$

Using the downward closure property, if an item set is infrequent, then its supersets are also infrequent. Accordingly, we can eliminate any set that includes Dog Food or Burgers (e.g. {Cat Food, Dog Food}, {Burgers, Vitamin Tablets}, {Burgers, Buns, Ketchup})

J. Which pairs of items could APriori probably eliminate if you knew more of the table?
We might be able to eliminate Vitamin C Tablets or Artisan Tap Water if more information was available. The item sets that contain these items have low purchase counts - it is likely that they are not popular and would be eliminated given more data,

K. Which pairs of items could APriori definitely not eliminate?

$$SUPP(CatFood) = 0.2 > 0.1$$
$$SUPP(CatLitter) = 0.14 > 0.1$$

Therefore, the supersets that contain Cat Food or Cat Litter (but not previously eliminated above) would remain and cannot be eliminated. For example, we cannot eliminate {Cat Food, Cat Litter}. We also don't know enough to eliminate {VitaminTablets, Artisan Tap Water}.

1.2 Neural Networks as Function Approximators

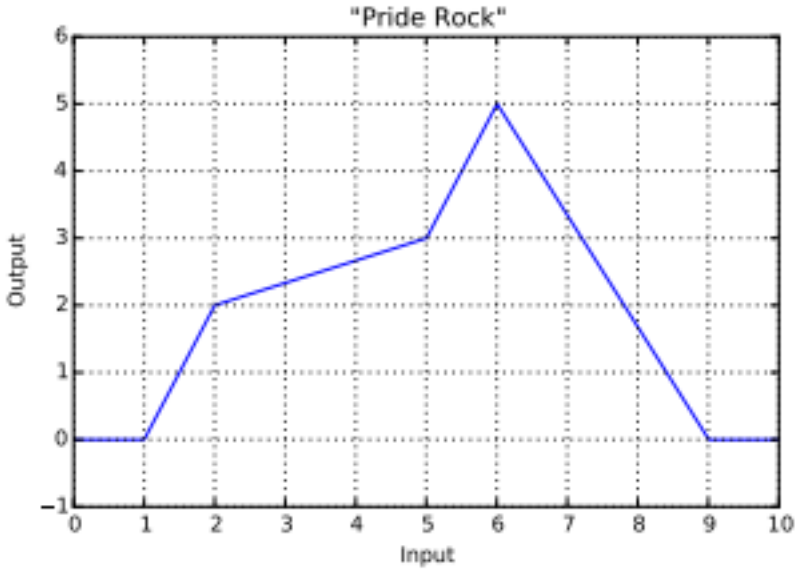
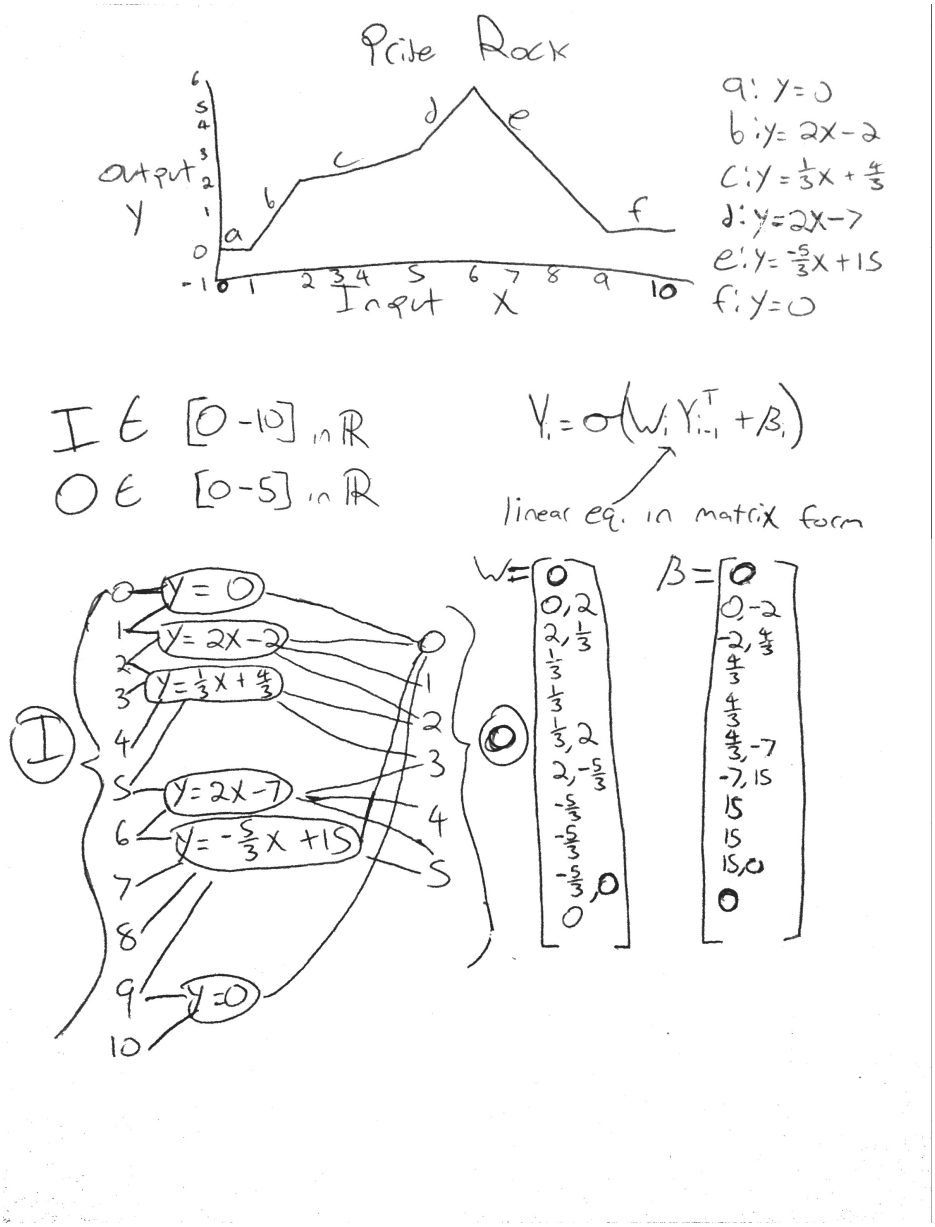


Figure 1: Example function to approximate using a neural network

NOTE: Since the graph is continuous, our input and output values can be any value within the range of the graph (not easily represented with the neural net, the connections below represent ranges of values).

Our feed-forward neural network consists of 1 input layer (allowing for input in the range $0 \leqslant I \leqslant 10$), 1 hidden layer with 6 hidden nodes (1 node per linear equation), and 1 output layer (within the range $0 \leqslant Output \leqslant 5$). The associated weights are the slope of each line (hidden node), and the biases are

the y-intercepts of those lines. Plugging in any input value within this range gives you an exact output 'prediction'. This allows us to produce the exact graph as shown above. Let's plot a few points as examples:
Input => Function => Output
$1.5 => 2(1.5) - 2 => 1$
$3.5 => \frac{1}{3}(3.5) + \frac{4}{3} => 2.5$
Now let's show attaining a vertex using two different hidden nodes/functions:
$6 => 2(6) - 7 => 5$
$6 => -\frac{5}{3}(6) + 15 => 5$



We could have used Relu to remove a single hidden node (the a and b above combined form a relu), as well as could have created pairs of nodes. This solution seemed cleanest.

1.3 Approximating Images with Neural Networks

A. The example network has 9 total layers: 1 input layer, 7 hidden layers and 1 output layer. The input layer declares the size of the input volume. The hidden layers ("fully connected" according to the documentation) perform the weighted addition of all inputs and pass along to the next layer. Ultimately, these activations reach the output layer which is a "regression layer." According to the documentation, this layer performs the backpropegation.

B. "Loss" refers to the output of a cost function - the lower the loss, the more accurate the neural network in approximating the image. The loss function implemented by this neural network (according to the GitHub source code) is an L2 cost function, or

$$\sum_{i=1} |x_i - y_i|^2$$

where x is the input and y is the user provided array of "correct" values.

C. We cloned the ConvNetJS GitHub repository and modified the source code of the "image regression" demo to display an additional graph plotting loss as a function of time. Analyzing the plot over 5000 iterations (using the cat image), the function converges to a value of approximately 0.0055.

The data was plotted in real-time to the canvas in the browser.

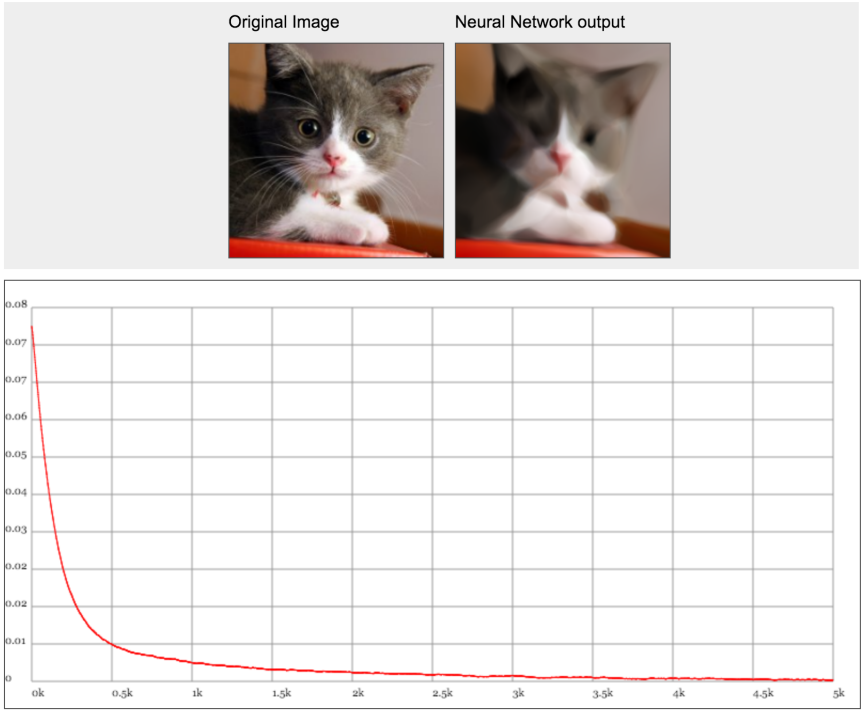


Figure 2: Loss as a function of time - 5000 iterations

D. By slightly modifying our previous function to plot loss vs. iterations, we managed to make the network converge to a lower loss function by lowering the learning rate by half every 1000 iterations. The resulting function converged to approximately 0.0035, lower than the previous example. These results were plotted in real-time in the browser and exported to an iPython Notebook for further analysis and comparison.

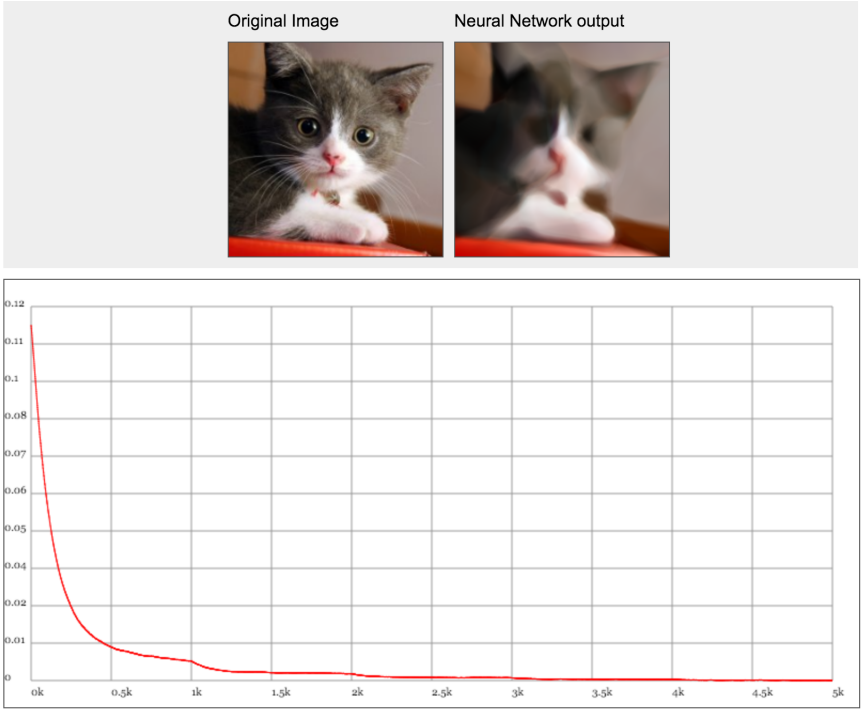


Figure 3: Loss as a function of time - learning rate halved every 1000 iterations

The data from Part C. and Part D. were plotted together on one graph for convenient comparison in Figure 4 below.

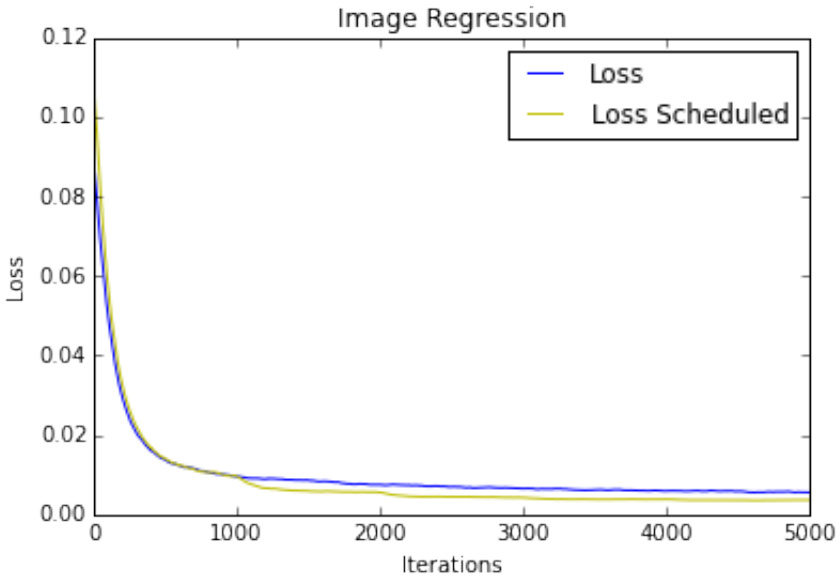


Figure 4: Loss as a function of time - comparison of Part C and Part D

E. By experimenting and dropping layer by layer, the loss function began to converge to a higher value with each layer deletion. While the original network converged to approximately 0.0035, that value rose by a significant digit to 0.010 by dropping 5 hidden layers (2 hidden layers remaining). Dropping 4 layers maintained a loss of approximately 0.0056.

   Therefore, the quality of the network by dropping layers is only noticeable when 5 or more layers are dropped.

F. By trial and error, we added additional hidden layers to the network and let it run for 5000 iterations. Despite the additional layers, the network seemed to converge to approximately the same value as previously (0.0035). We added up to 30 layers and the network accuracy still seemed to remain stable.

2 Part 2: Association Rule Learning

2.1 Summary

Association analysis is the task of finding interesting relationships in large data sets. In this assignment, we were tasked to perform association analysis on Project VoteSmart data to determine association rules in voting behavior by politicians.

2.2 Solution Steps

A. Implementation - The data set was first properly parsed into transaction entries. Each row represents a politician and each corresponding column value represents a mapping to a bill vote or party affiliation (e.g. Republican). A "meaning" dictionary was constructed to map the keys to their meanings.

   Next, we adapted the code from Harrington Sec.11.5 and included proper reference citation. The implementation trains using the Apriori algorithm to find frequent item sets and support data. The implementation also includes methods to generate rules from this data (see below).

B. Frequent Item Sets - The default minimum support level was set to 50% and decremented in intervals of 5% until reaching 30%. The item set output and counts are displayed in below (Fig. 5).

```
for threshold in np.arange(0.5, 0.25, -0.05):
    itemsets, support = apriori.apriori(transactions.values(), minSupport=threshold)
    print "THRESHOLD: ", threshold
    print len(itemsets), "itemsets of length:"
    print [len(i) for i in itemsets]
    print "\n"

THRESHOLD: 0.5
3 itemsets of length:
[4, 1, 0]


THRESHOLD: 0.45
3 itemsets of length:
[9, 2, 0]


THRESHOLD: 0.4
4 itemsets of length:
[14, 24, 9, 0]


THRESHOLD: 0.35
7 itemsets of length:
[15, 48, 68, 48, 16, 2, 0]


THRESHOLD: 0.3
8 itemsets of length:
[21, 69, 104, 108, 77, 32, 6, 0]
```

Figure 5: Frequent Item Sets and Counts

C. Association Rules -

(a) Using the threshold of 30% above, we performed association rule mining with minimum confidence ranging from 70% to 99% with 5% increments. Sample results are displayed in Fig. 6.

(b) The generated rules from the previously calculated itemsets and support data were next mapped to their English meaning. Fig. 7 demonstrates six sample rules learned from the above process and the corresponding confidence levels of each rule.

```
itemset, support = apriori.apriori(transactions.values(), minSupport=0.3)
for threshold in np.arange(0.7, 0.99, 0.05):
    print "THRESHOLD: ", threshold
    rules = apriori.generateRules(itemset, support, minConf=threshold)
    print "\n"

THRESHOLD: 0.7
frozenset([15]) --> frozenset([1]) conf: 0.961538461538
frozenset([1]) --> frozenset([15]) conf: 0.827205882353
frozenset([15]) --> frozenset([21]) conf: 0.897435897436
frozenset([22]) --> frozenset([1]) conf: 0.951351351351
frozenset([3]) --> frozenset([25]) conf: 0.94298245614
frozenset([25]) --> frozenset([3]) conf: 0.80223880597
frozenset([3]) --> frozenset([26]) conf: 0.872807017544
frozenset([26]) --> frozenset([3]) conf: 0.708185053381
frozenset([17]) --> frozenset([1]) conf: 0.87037037037
frozenset([1]) --> frozenset([17]) conf: 0.863970588235
frozenset([29]) --> frozenset([1]) conf: 0.981308411215
...
```

Figure 6: Frequent Item Sets and Counts

```
IF: 'Mine Safety Act -- Yea' AND 'Prohibiting 2010- 2011 Congressional
Cost-of-Living Pay Increase -- Yea' AND 'Repealing "Don\\\'t Ask,
Don\\\'t Tell" After Military Review and Certification -- Yea'
THEN: 'Democratic' AND 'Unemployment Benefits Extension -- Yea'
CONFIDENCE: 0.961111111111


IF: 'Republican' AND 'Removing Troops from Afghanistan -- Nay' AND
'Prohibiting Use of Federal Funds For Planned Parenthood -- Yea'
THEN: 'Repealing the Health Care Bill -- Yea' AND 'Terminating the
Home Affordable Modification Program -- Yea'
CONFIDENCE: 0.981220657277


IF: 'Prohibiting Federal Funding of National Public Radio -- Yea' AND
'Removing Troops from Afghanistan -- Nay'
THEN: 'Repealing the Health Care Bill -- Yea' AND 'Terminating the
Home Affordable Modification Program -- Yea'
CONFIDENCE: 0.976744186047


IF: 'Republican' AND 'Reducing Federal Funding of the US Institute of
Peace -- Yea' AND 'Prohibiting Federal Funding of National Public
Radio -- Yea' AND 'Removing Troops from Afghanistan -- Nay'
THEN: 'Repealing the Health Care Bill -- Yea' AND 'Terminating the
Home Affordable Modification Program -- Yea'
CONFIDENCE: 0.975124378109


IF: 'Repealing the Health Care Bill -- Yea' AND 'Prohibiting Federal
Funding of National Public Radio -- Yea' AND 'Reducing Federal
Funding of the US Institute of Peace -- Yea'
THEN: 'Republican' AND 'Prohibiting Use of Federal Funds For Planned Parenthood -- Yea'
CONFIDENCE: 0.958139534884


IF: 'Repealing the Health Care Bill -- Yea' AND 'Prohibiting Federal
Funding of National Public Radio -- Yea' AND 'Removing Troops from
Afghanistan -- Nay' AND 'Reducing Federal Funding of the US Institute of Peace -- Yea'
THEN: 'Republican' AND 'Prohibiting Use of Federal Funds For Planned Parenthood -- Yea'
CONFIDENCE: 0.960396039604
```

Figure 7: Meaning of Discovered Rules and Confidence Levels

3 Part 3: Random Forest For Image Approximation

A. For our image approximation using random forest, we chose to use the Mona Lisa.

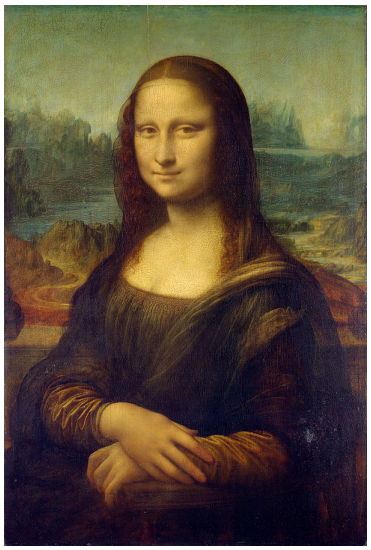


Figure 8: Mona Lisa

B. In general, little if any preprocessing is done to data that uses random forests. There is no need to perform mean subtraction or standardization in this case, since we're not calculating any distances between pixels. Below is how we attained our "training set" of 5,000 randomly sampled (x, y) coordinates within the image.

```
width, height = mona.size

TRAIN_COORDINATES = []
while len(TRAIN_COORDINATES) < 5000:
  rand_point = (np.random.randint(width), np.random.randint(height))
  if not rand_point in TRAIN_COORDINATES:
    TRAIN_COORDINATES.append(rand_point)
```

Figure 9: Sampling of 5,000 Coordinates

C. There are many different ways of handling the preprocessing of these output pixels, but we decided on a built in weighted grayscale conversion ($\tilde{R}$ * 0.299 + G * 0.587 + B * 0.114). The "target" values from the training coordinates were found by taking that coordinate's grayscale pixel value.

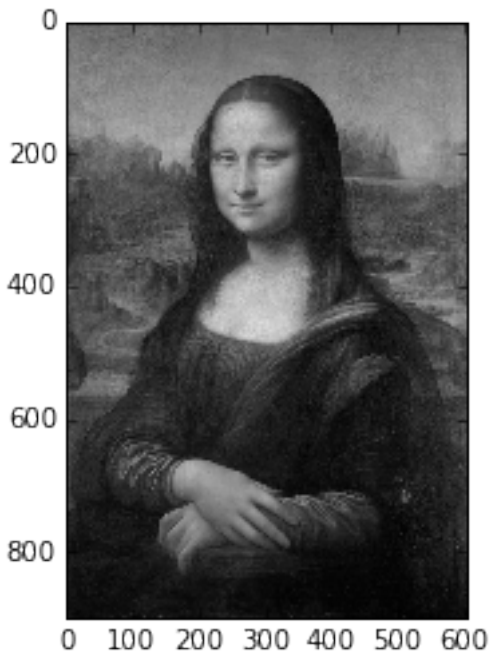


Figure 10: Mona Lisa Grayscale

```
gray_mona = mona.convert('LA')
plt.imshow(gray_mona)

mona_pixels_gray = gray_mona.load()
POST_PIXELS = []
for pixel in TRAIN_COORDINATES:
  grayscale, pixel_range = mona_pixels_gray[pixel[0], pixel[1]]
  POST_PIXELS.append(grayscale)
```

Figure 11: Attaining "Target" Post Pixels

D. Pixel re-scaling ended up being unnecessary, however the scaling code can be seen below.

```
RESCALED_PIXELS = np.array(POST_PIXELS) / 255
```

Figure 12: Pixel Re-scaling

E. As mentioned above, preprocessing isn't common for random regression forests, and in this case seems unnecessary. Therefore, there were no other implemented pre-processing steps.

F. To build our final image, we trained a 10 decision tree forest with no max depth. To train the random forest, we pass in the training coordinates,

and their corresponding grayscale pixel brightness/value. The test data is all coordinates within the bounds of the image size, i.e.
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (0, 10), (0, 11), (0, 12), (0, 13), (0, 14), (0, 15), (0, 16), (0, 17), (0, 18), (0, 19), (0, 20), (0, 21), (0, 22), (0, 23), (0, 24), (0, 25), (0, 26), (0, 27), (0, 28), (0, 29), (0, 30), (0, 31), (0, 32), ...(1,0), (1, 1), ..., (width, height)]
The resulting prediction image and code that generated it can be found below.

```
# build the test data (all coordinates corresponding to the size of the image)
TEST = [(y, x) for x in range(height) for y in range(width)]

def rand_forest(num_trees, depth):
  rf = RF(n_estimators=num_trees, max_depth=depth)
  rf.fit(TRAIN_COORDINATES, POST_PIXELS)
  prediction = rf.predict(TEST)
  a = np.array(prediction).reshape(height, width)
  print num_trees, " decision trees with max depth ", depth
  plt.imshow(a, cmap="gray")
  plt.show()

rand_forest(10, None)
```

Figure 13: Random Forest Implementation

G. 3.1 Experimentation

(a) We repeated random forests using a single decision tree, but in our case with depths 1, 2, 3, 5, 10, 15. In the range of 1-15, the greater the depth, the better the result. This is not always the case, as overfitting is a possibility. The "best" depth can be tested with cross validation. In general, depth is the number of random subsets of features that the random forests trees will consider when splitting a node. The smaller the depth, the smaller the variance, but the higher the bias. The general "default" is the number of features. The resulting images from the variety of depths can be found below.

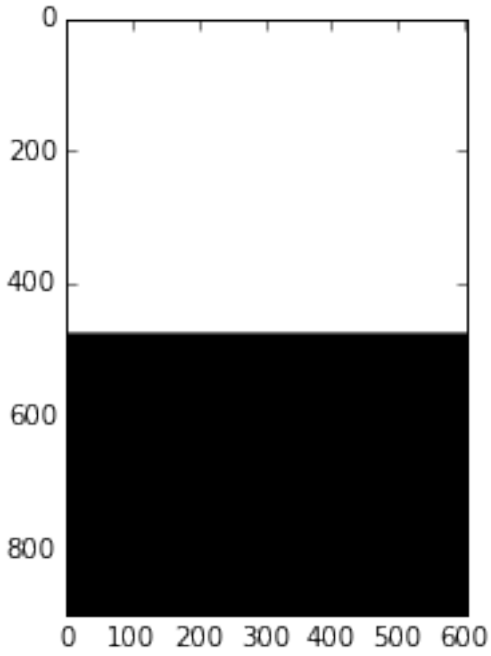


Figure 14: 1 Tree Using Depth = 1

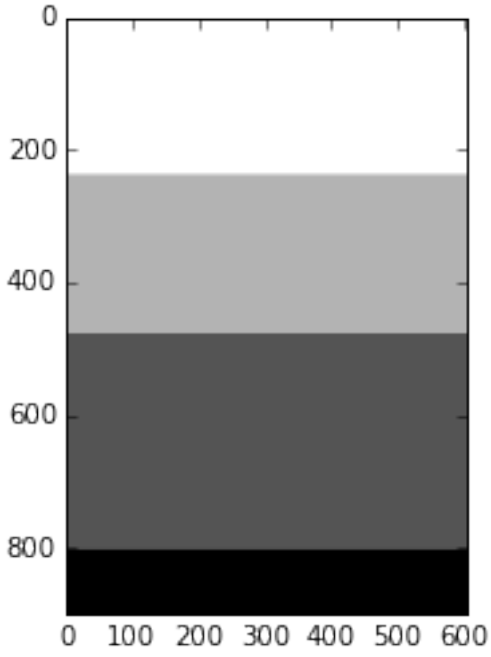


Figure 15: 1 Tree Using Depth = 2

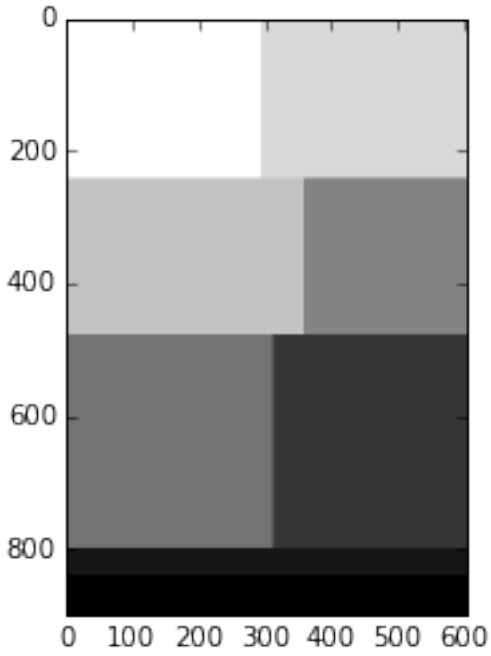


Figure 16: 1 Tree Using Depth = 3

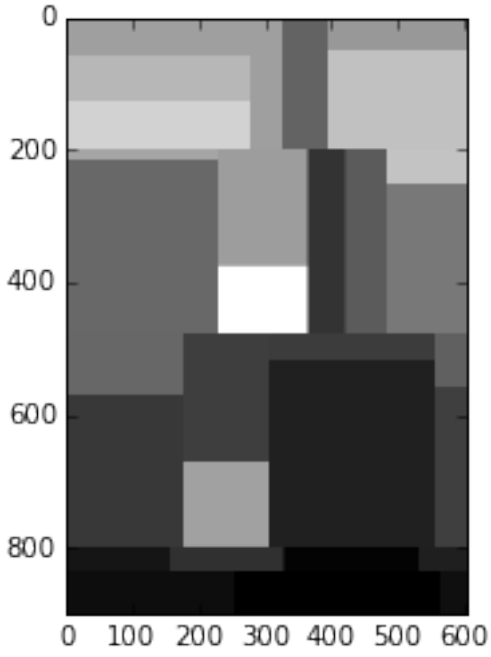


Figure 17: 1 Tree Using Depth = 5

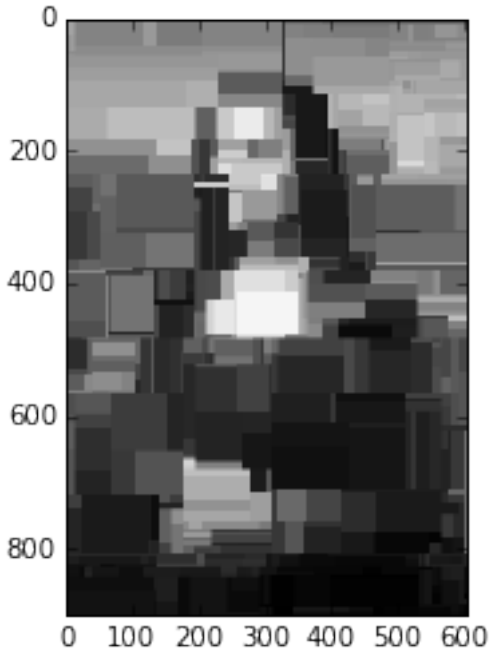


Figure 18: 1 Tree Using Depth = 10

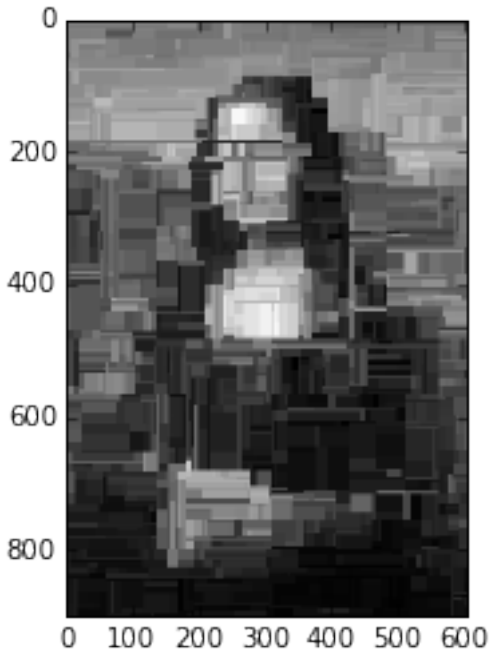


Figure 19: 1 Tree Using Depth = 15

(b) We repeated the experiment with different numbers of trees in the

random forest. We used 1, 3, 5, 10, 100 trees, each with a depth of 7. As the number of trees grew, the image prediction became more accurate, however there is a cut off point. The higher the trees, the longer the regression takes, and at a certain number, there is little to no distinguishable difference between the results as the number of trees increases. For example, using our case, there is a very slight difference between 10 and 100 trees.

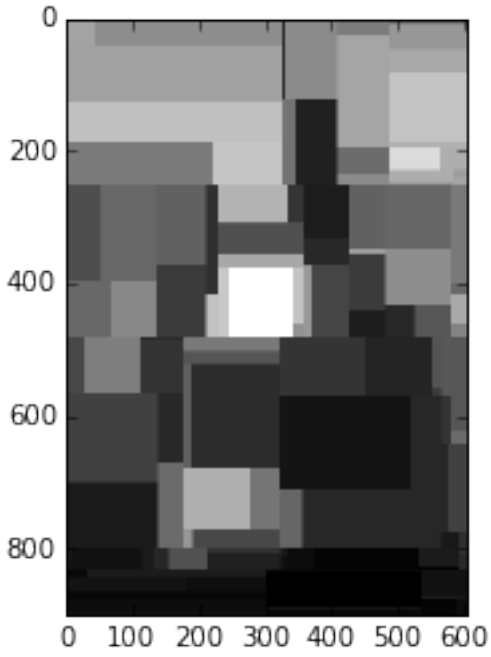


Figure 20: 1 Tree Using Depth = 7

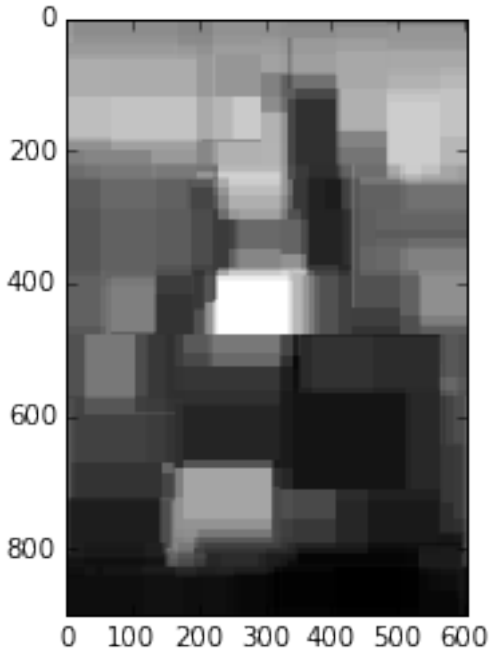


Figure 21: 3 Trees Using Depth = 7

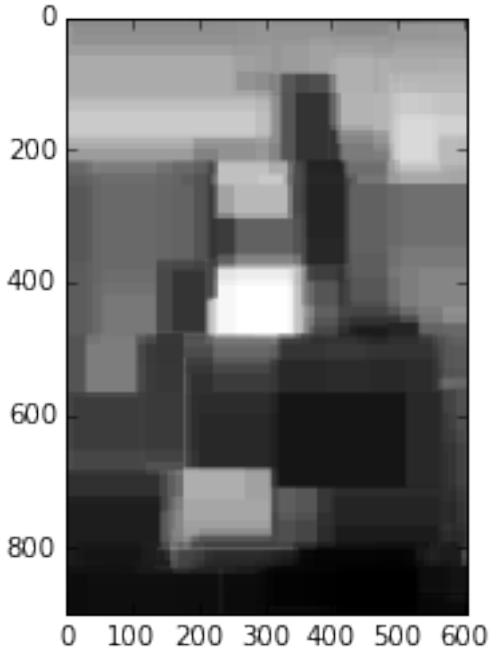


Figure 22: 5 Trees Using Depth = 7

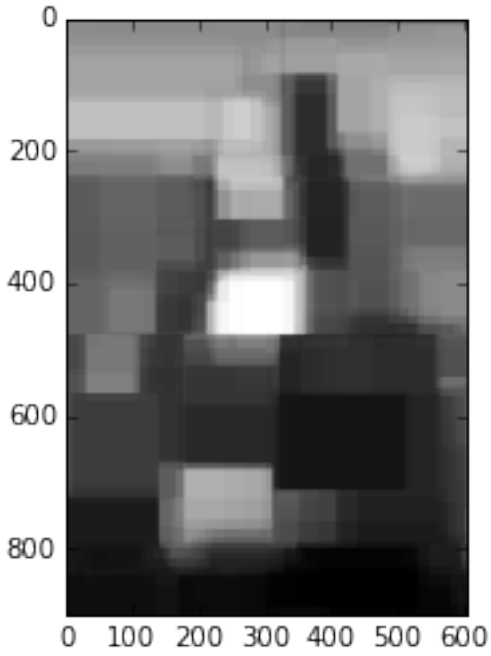


Figure 23: 10 Trees Using Depth = 7

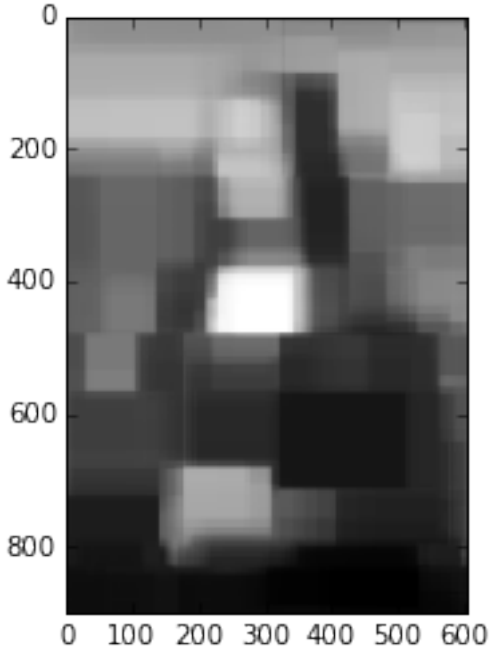


Figure 24: 100 Trees Using Depth = 7

(c) After pruning the depth and number of trees, we repeated the ex-

periment using a k = 1 k-NN regressor. This means that every pixel in the output equals the nearest pixel from the training set. This created a more "pixelated" image, rather than the "patchy" images attained from the random forest regressors. The reason for the "pixelated" batches is because we're assigning the nearest pixel colors (k = 1) to these coordinates, and we assume (as does the regressor) that coordinates within close proximity to one another will have a similar pixel brightness. This being said, little clusters of pixels are assigned to the same pixel brightness, causing a "pixelated" output.

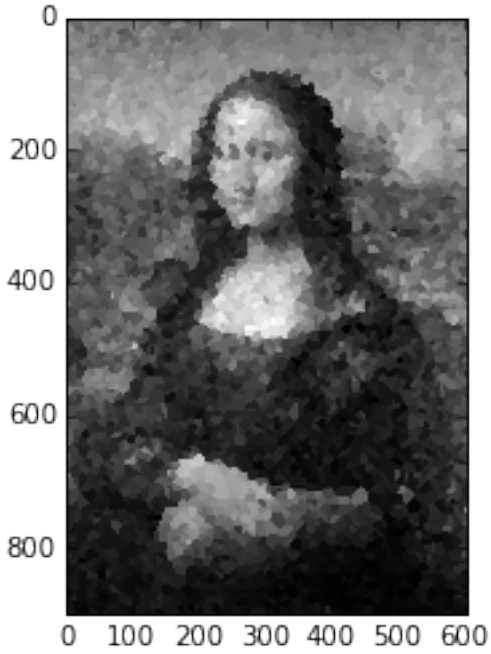


Figure 25: k = 1 k-NN Regressor

(d) After pruning and cross validating to attain the "best" values, we came to the conclusion that 100 trees with NO max depth and 1000 trees with 100 max depth were the most consistently accurate combinations. Those were our cut-off points, and we did not see much, if any, positive improvement as the trees exceeded 100 or 1000. Below you can find one trial run scores of combinations using the notation (trees, depth).

```
# cross validation, k = 10 for pruning
prune_trees = [10, 25, 50, 100, 1000]
prune_depth = [10, 25, 50, 100, None]
best_accuracy = -1
best_prune = (-1, -1)

for tree in prune_trees:
    for depth in prune_depth:
        rf = RF(n_estimators=tree, max_depth=depth)
        score = cv.cross_val_score(rf, TRAIN_COORDINATES, POST_PIXELS, cv=10).mean()
        print (tree, depth), score
        if score > best_accuracy:
            best_accuracy = score
            best_prune = (tree, depth)

print "Highest Consistent Accuracy: {0} with {1}".format(best_accuracy, best_prune)

(10, 10) 0.886828662262
(10, 25) 0.889429142508
(10, 50) 0.890753117534
(10, 100) 0.890051881492
(10, None) 0.891398920508
(25, 10) 0.893409488332
(25, 25) 0.896301706423
(25, 50) 0.896618066527
(25, 100) 0.895815651292
(25, None) 0.895301428687
(50, 10) 0.893913713826
(50, 25) 0.897063485176
(50, 50) 0.898906407502
(50, 100) 0.897935975529
(50, None) 0.897440344671
(100, 10) 0.894843526667
(100, 25) 0.898690400422
(100, 50) 0.900045323044
(100, 100) 0.899547337079
(100, None) 0.900484385304
(1000, 10) 0.895832591216
(1000, 25) 0.900138924699
(1000, 50) 0.900093057642
(1000, 100) 0.900286179601
(1000, None) 0.900307097712

Highest Consistent Accuracy: 0.900484385304 with (100, None)
```

Figure 26: Pruning and Cross Validation

3.2 Analysis

(a) The decision rule at each split point is made based on whether we are or aren't in the range of a specific (current) row or column within the image. For example, we'll use the case of the tree with a max

depth of 1. If we take a closer inspection of the grayscale mona lisa image, the human eye can easily detect that the upper portion is lighter than the bottom portion. (continue reading below)

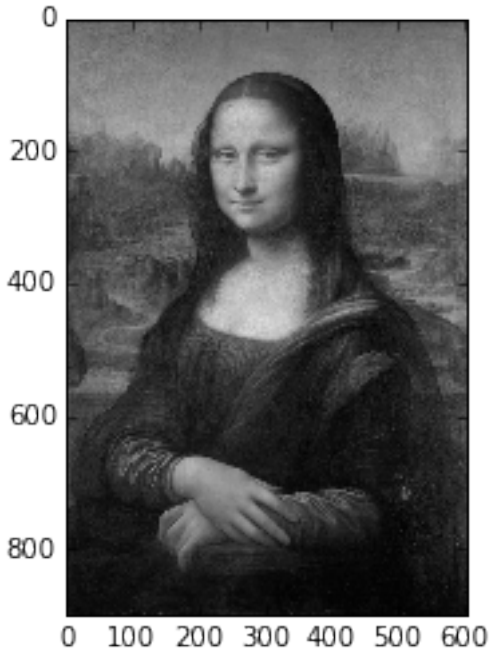


Figure 27: Mona Lisa Grayscale

Since we're only dealing with a depth of 2, we're dealing with 2 colors. The upper portion of the image will be set to white (since it's brighter), and the lower portion will be set black. This decision rule is reflected in the following images.

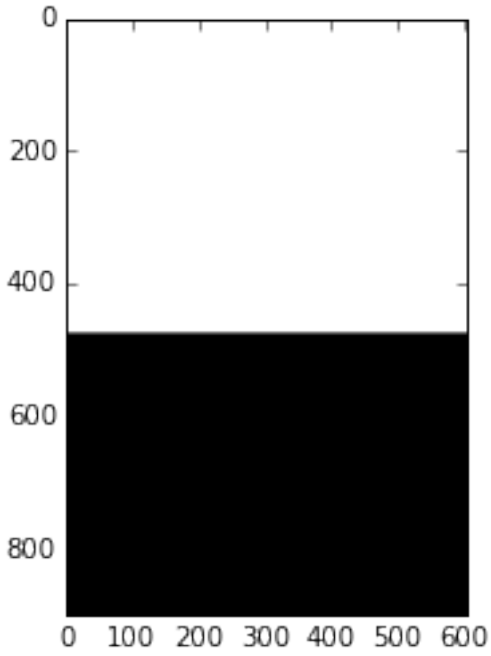
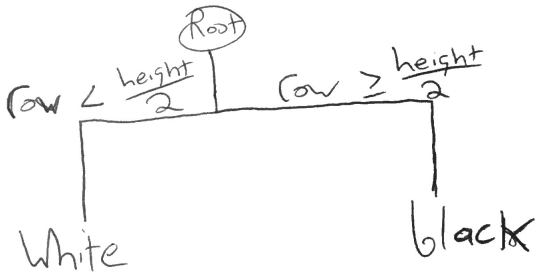
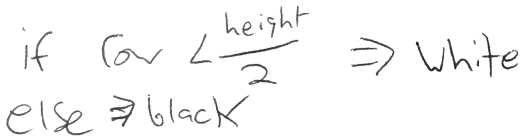
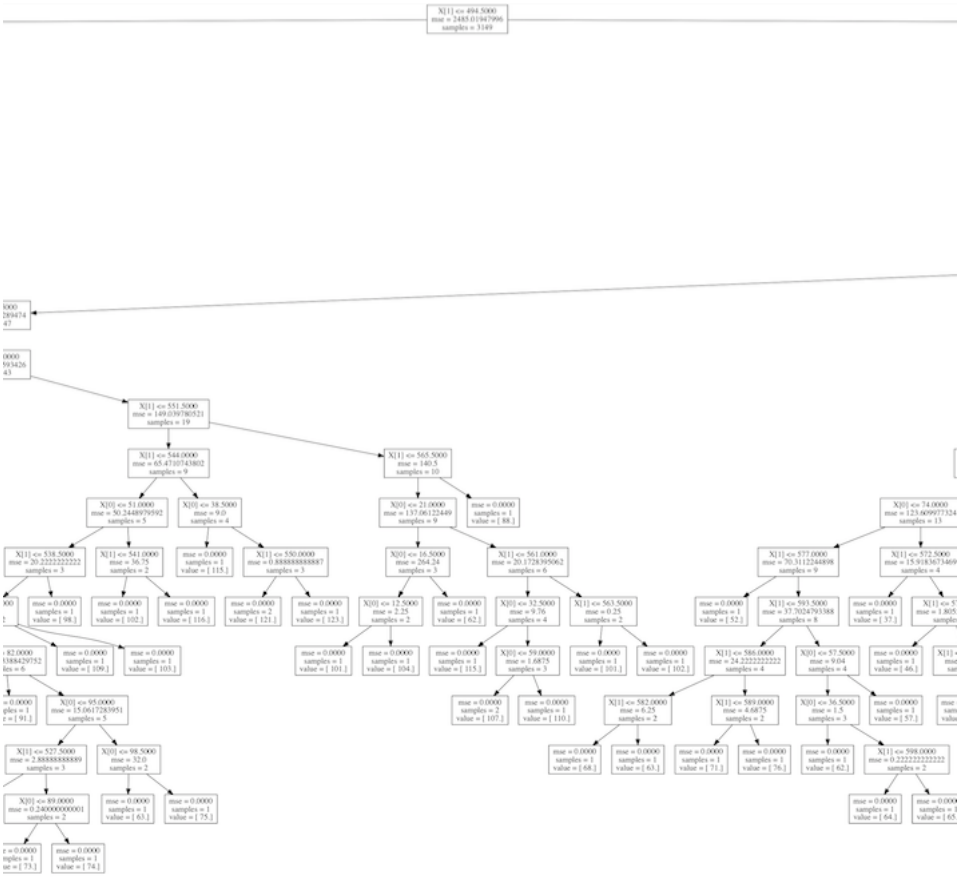


Figure 28: 1 Tree Using Depth = 1





Here is a small sample portion from a random decision tree (using our rf regressor), and the code that generated it. The entire tree can be found in the .zip folder.



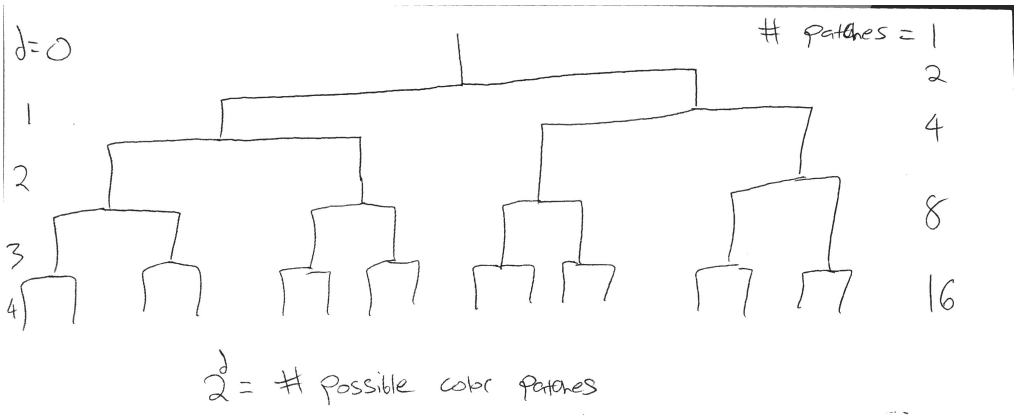
```
# saving random decision tree
rf = RF(n_estimators=100, max_depth=None)
rf.fit(TRAIN_COORDINATES, POST_PIXELS)
with open('decision_tree.dot', 'w') as my_file:
   my_file = tree.export_graphviz(rf.estimators_[0], out_file = my_file)
```

Figure 29: Sample Decision Tree

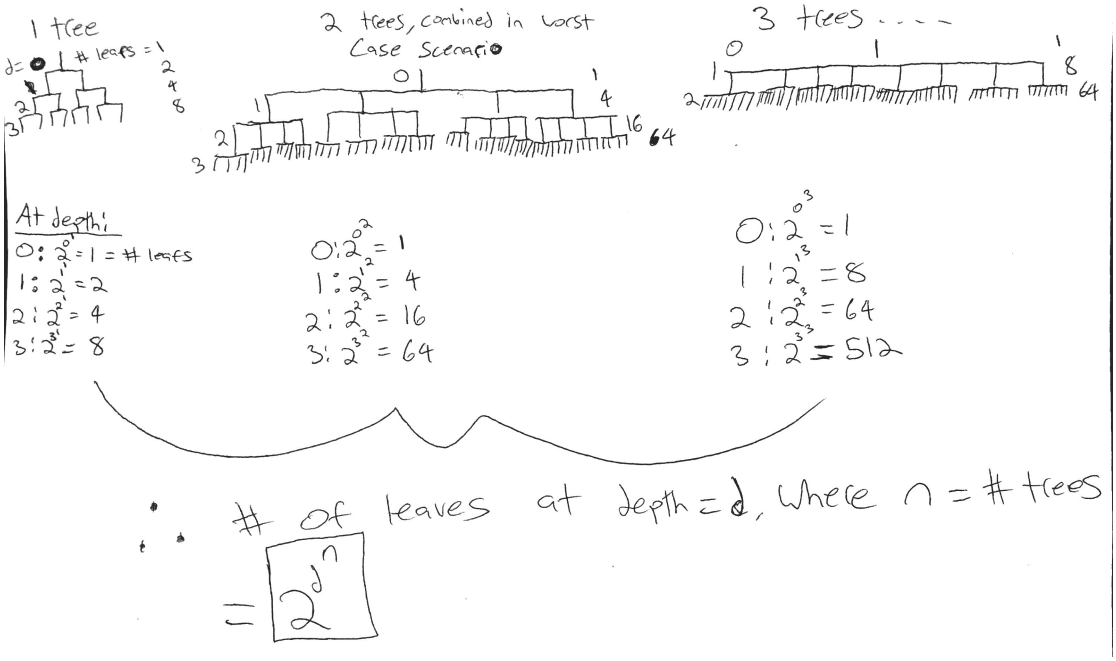
(b) The resulting image is comprised of rectangular "patches" which stem from the partitions created by the decision trees in our random forest regressor. The greater the tree depth, the more patches, and the greater number of trees, the smoother and more accurate the patches are. The organization of the patches is similar to a look-up table: for example, the color of a specific gray in a region/patch is the predicted pixel brightness (determined after training) mapped based on the coordinates within its range.

(c) The number of patches of colors in a resulting image is completely

dependent upon the number of leaves at a given depth of a tree. Given that depth = d, we can show (see below) that there may be up to 2^d different color patches.



(d) The number of patches of color that might be in a resulting image if the forest contains n decision trees falls in quite a large range. The minimum number of patches of color that might be in the resulting image is the same number as if we'd use 1 tree, i.e. 2^d. This is a rare case, but theoretically there can be identical trees, causing perfect overlap (i.e. all the same trees and decisions), and the n in a "minimal" case $n * 2^d$ (where n is the number of trees) can therefore be treated as 1, leaving us with a lower bound of just 2^d. The upper bound is slightly more complicated. We see that if the trees are entirely different (worst case scenario), there are no decisions in common. This means that we can treat n trees as a single 2^n-ary tree. For example, if n = 1, we have a simple case of a 2-ary tree, known as a binary tree. In the case of n = 3, we have a case of a 2^3-ary tree. To get the number of leaf nodes, or colors in this case of a specific depth = d in a 2^n-ary tree is equal to: 2^{d^n}. The image below shows slightly more in depth as to how this works.


1 tree
leaves = 1
2
4
8
2 trees, combined in worst Case Scenario
3 trees . . . .
At depth:
0: $2^0 = 1$ = # leaves
1: $2^1 = 2$
2: $2^2 = 4$
3: $2^3 = 8$
0: $2^0 = 1$
1: $2^2 = 4$
2: $2^4 = 16$
3: $2^6 = 64$
0: $2^0 = 1$
1: $2^3 = 8$
2: $2^6 = 64$
3: $2^9 = 512$
∴ # of leaves at depth = d, where n = # trees
$= 2^{dn}$

References

[1] HTF: The Elements of Statistical Learning
http://statweb.stanford.edu/ tibs/ElemStatLearn/

[2] Matplotlib Plotting (Histograms, pixel images, etc):
http://matplotlib.org/

[3] Numpy
http://www.numpy.org/

[4] Pandas Data Parser (read_csv, to_csv):
http://pandas.pydata.org/

[5] Scikit-learn
http://scikit-learn.org/

[6] Scipy
http://www.scipy.org/

[7] Exporting Decision Tree in Sklearn
https://biesnecker.com/2015/03/27/visualizing-a-scikit-learn-decision-tree/