HW 4: Association Rule Learning, Neural Nets, Neural Networks, Random Forests

Kiyan Rajabi (kr499) & Roy Cohen (rc726)
CS 5785, Applied Machine Learning
Date: 23 November 2016

CORNELL TECH

Abstract

This assignment allows us to get exposure to testing various parameters of the following concepts: Association Rule Learning, Neural Networks, and Random Forests.

1. Motivation

Association Rule Learning helps us discover relationships between variables in large databases. It was originally used to discover regularities among items purchased at the supermarket, and that's the way in which we use it in our analysis for this exercise.

Neural Networks are based on neural units that have a summation function that combines the inputs together. These are processed in hidden layers and communicated from the input layers, and they then link to a resultant output layer.

Random forests is a regression and classification algorithm generated from decision trees. It calculates a response variable by creating these decision trees, and consequently putting each object down the trees.

These methods are useful in solving real-world problems, some of which are demonstrated in the following report.

2. Material and Methods

We used the following online resources and coding platforms:

- iPython Notebook
- Libraries used:
    - Numpy
    - Matplotlib

3. Results

3.1 Association Rule Learning

For the first part of the assignment, we analyzed consumer spending habits of customers of Nile.com. This required us inspecting purchase combinations and using association rules to determine whether particular combinations can be profitable.

For the following problem set, we provided upper and lower bounds

1. Written Exercises
Association Rule Learning

Total a) Nile.com made 927,125 purchases last month

vit C + other b) Lower bound: 231 customers
Upper Bound: 927,125 (all) − (29,751 (burgers) − 231 (burgers + vit C)) = 897,605

Cat food only c) Lower: 185,279 − (60,159 + 120,091) = 5,029
Upper: 185,279 (cat food) − 120,091 (cat food and ...) = 65,188

d) Supp(Burgers, Vit C, Ketchup)
↳ frequency of itemsets
Lower: 0% => can't be certain these purchased together
Upper: 231 => Burgers, vit C / 927,125 -> all items => 0.025 %

e) Supp(Burgers, Buns)
Lower: 15,293 => Burgers, Buns, and Ketchup / 927,125 => all items
=> 1.65 %
Upper: 29,751 => Burgers / 927,125 => all => 3.21 %

f) Conf({Burgers}) => {Vit C}
↳ indication of how often rule found to be true
↳ proportion of transactions that contain X which also contain Y
↳ conf(B → VitC) = Supp(B ∪ VitC) / Sup(B)

(B, VitC) → 231 / 927,125 ÷ 29,751 / 927,125 ← (Burgers)
↑ all ↑ all

= 0.78% => very low confidence, so this is not a good promotion

g) Conf({Dog Food, Cat Food} → {Cat Litter}

=> Supp(DF ∪ CF ∪ CL) / supp(DF ∪ CF)

Lower = 0/60,159 => 0%

Upper = 60,159 / 60,159 = 100%
↑ CF, DF ↑ CF, DF

H) Lift (Dog Food → Cat Food)

↳ $\frac{\text{supp}(DF \cup CF)}{\text{supp}(DF) \cdot \text{supp}(CF)}$

$= \frac{(60,159 / 927,125)}{\left(\frac{80,915}{927,125}\right) \cdot \left(\frac{185,279}{927,125}\right)} = 3.72$

{CF, DF}, all (DF) (all) (CF) (all)

I) APriori => uses bottom up

↳ min support of 0.1

$\text{SUPP(Burgers)} = \frac{29,751}{927,125} = 0.032$ | $\text{Supp(DF)} = \frac{80,915}{927,125} = 0.087$

of these, eliminate sets that contain either. Thus:

{Dog Food}, {Cat Food, Dog Food}, {Burgers}, {Burgers, Vit C}, {Burgers, Buns, Ketchup}

J) Other items we might be able to remove:

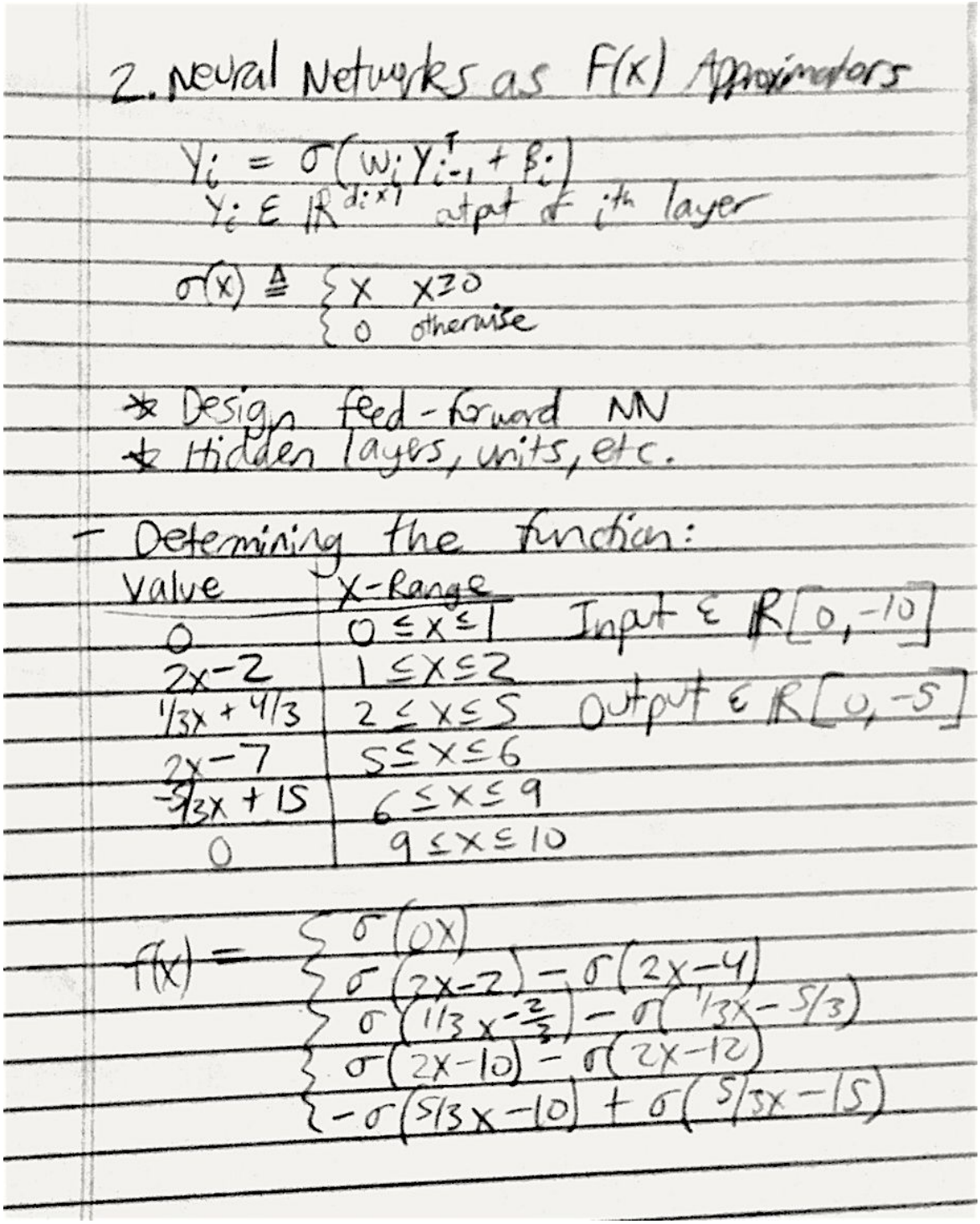
{Vitamin C}, {Artisan Water}, {Buns}, {Ketchup}

K) Definitely not eliminate:

$\text{Supp(CL)} = \frac{130,122}{927,125} = 0.14$; $\text{Supp(CF)} = \frac{185,279}{927,125} = 0.20$

{Cat Food}, {Cat Litter}; {Cat Food, Cat Litter}

3.2 Neural Networks as Function Approximators



2. Neural Networks as F(x) Approximators

$Y_i = \sigma(W_i Y_{i-1}^T + B_i)$

$Y_i \in \mathbb{R}^{d_i \times 1}$ output of i^{th} layer

$\sigma(x) \triangleq \begin{cases} x & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$

* Design feed-forward NN
* Hidden layers, units, etc.

- Determining the function:

Value	X-Range
0	$0 \leq x \leq 1$
$2x-2$	$1 \leq x \leq 2$
$1/3x + 4/3$	$2 \leq x \leq 5$
$2x-7$	$5 \leq x \leq 6$
$-5/3x + 15$	$6 \leq x \leq 9$
0	$9 \leq x \leq 10$

Input $\in \mathbb{R}[0, -10]$

Output $\in \mathbb{R}[0, -5]$

$$f(x) = \begin{cases} \sigma(0x) \\ \sigma(2x-2) - \sigma(2x-4) \\ \sigma(1/3x - 2/3) - \sigma(1/3x - 5/3) \\ \sigma(2x-10) - \sigma(2x-12) \\ -\sigma(5/3x-10) + \sigma(5/3x-15) \end{cases}$$

As you can see from the generation of our neural network, we assign five rectified linear units. There is only one hidden layer required. After, we assigned the respective weights and bias' that would help us generate output values from the function.

```
#define the input
n=np.linspace(0,10,100)
```

```
#assign weights
weight=np.array([2,float(5)/3,float(5)/3,float(11)/3,float(5)/3])
weight1=np.array([1,-1,1,-1,1])
```

```
#assign bias
bias=np.array([-2,float(-10)/3,float(-25)/3,-22,-15])
bias1=0
```

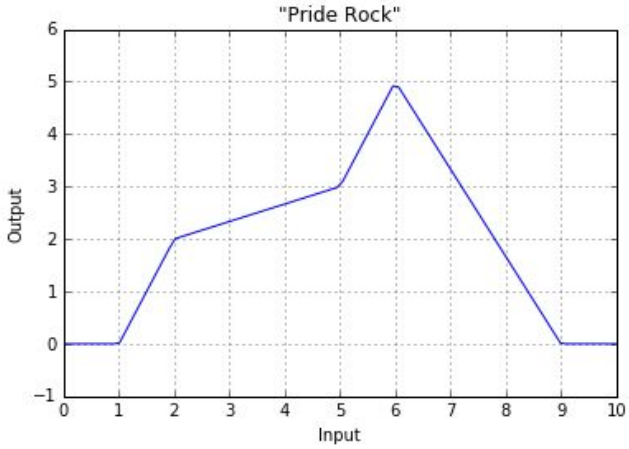
```
#max f(x)
def nnFA(n):
    return np.maximum(n,0)
```

```
vals=[]
```

```
for i in range(len(n)):
    fin=np.array(n[i]).dot(weight)+bias
    sec=nnFA(fin)
    sec2=sec.dot(weight1)+bias1
    vals.append(sec2)
```

Our results were then plotted to assimilate the plot that was provided:

```
plt.title('"Pride Rock"')
plt.xlabel('Input')
plt.ylabel('Output')
plt.grid(True)
plt.plot(n,vals)
plt.axis([0, 10, -1, 6])
plt.xticks(range(0,11))
plt.show()
```



3.3 Approximating Images with Neural Networks

In this section, we explore a convolutional neural network package implemented in JavaScript.

ConvNetJS takes (x,y) position on a grid and learns to predict the color at that point using regression to (r,g,b).

3.3.1 Structure of the Network

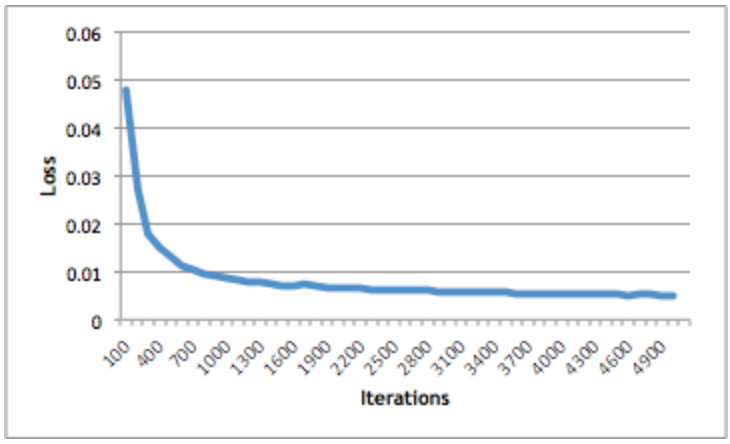
There are layers total. The initial layer is the input layer comprised of x,y. The following layers are the hidden, fully-connected layers comprised of 20 neurons, as well as reLU activation. The final output layer is a regression layer that predicts three r,g,b output neurons.

3.3.2 Loss Definition

According to the source code in GitHub, loss is the class negative log likelihood. As a result, here is the loss function for this neural network:

```
// implements an L2 regression cost layer,
// so penalizes \sum_i(||x_i - y_i||^2), where x is its input
// and y is the user-provided array of "correct" values.
var RegressionLayer = function(opt) {
  var opt = opt || {};
```

3.3.1 Plotted Loss Over Time



As you can see from the plot, we are at about a 0.005 loss after 5,000 iterations. In addition, it is visible that there is a drop in loss after the first 100 iterations.

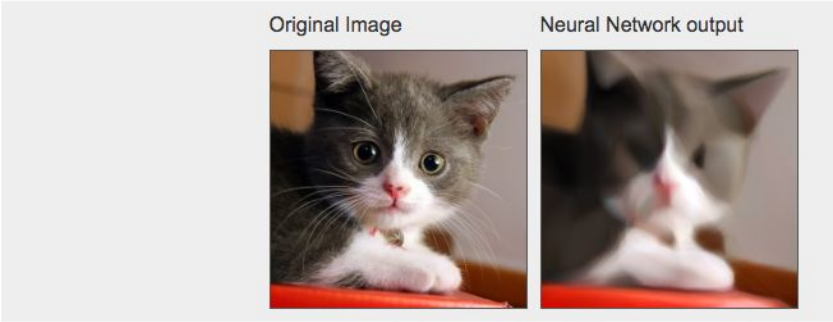
3.3.1 Network Convergence

Yes! By lowering the learning rate every 1,000 iterations, we can make the network converge to a lower loss function. By lowering the learning rate, you also effectively reduce loss as well.

3.3.1 Lesion Study

We experimented with the omission of a number of layers and determined that 5 layers is approximately the threshold that starts resulting in a noticeable difference in loss. As you can see below, after removing 5 layers the loss of 0.0129 is still quite high at 5,000 iterations. You can also make out the outline from the image, but if you continue hiding more layers than that, the image becomes increasingly difficult to decipher.

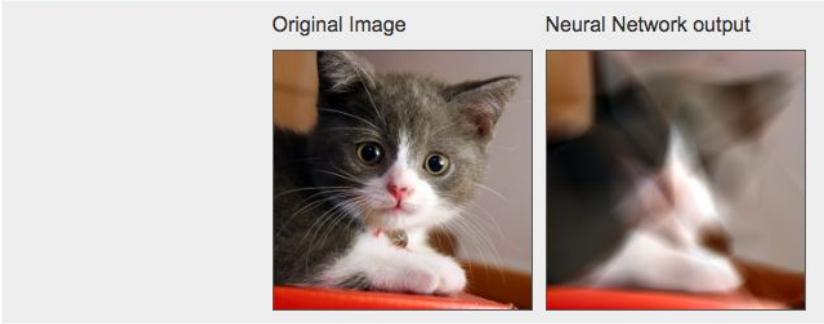
Original (no layers hidden):



loss: 0.005697486967359673
iteration: 4999

Learning rate: 0.1

After removing 4 layers, the loss was 0.0073 after 5,000 iterations.



loss: 0.006371616405769162
iteration: 4999

Learning rate: 0.01

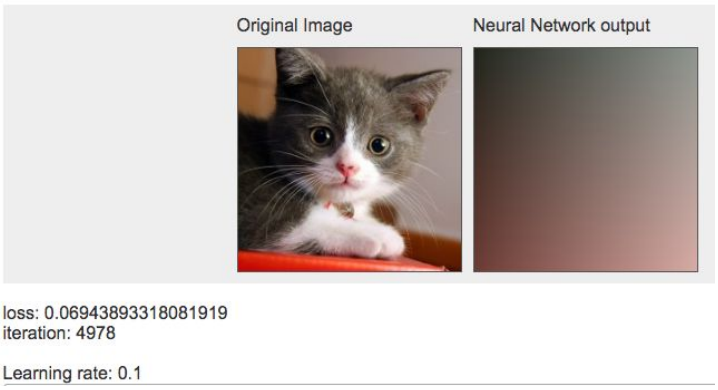
After removing 5 layers, the loss was 0.0129 after 5,000 iterations:

loss: 0.012970772506607567
iteration: 4999

Learning rate: 0.1

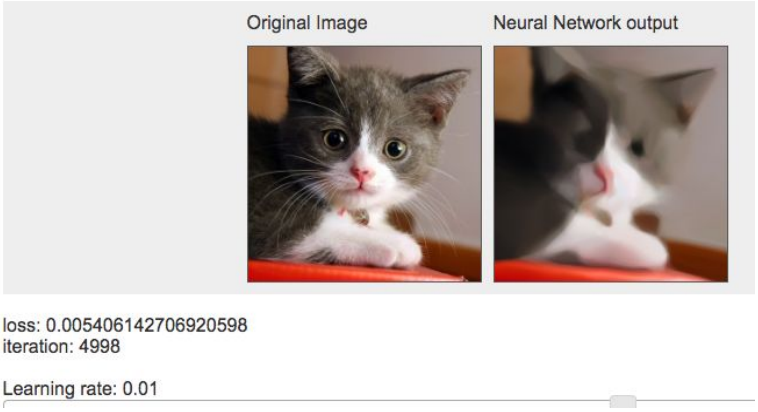
After removing 6 layers, the loss was 0.0267 after 5,000 iterations:

After removing all 7 hidden layers, the loss was still about 0.069 after 5,000 iterations:



3.3.1 Improving Accuracy

We added a couple of extra hidden layers but did not notice a noticeable accuracy improvement. It was actually approximately the same as with the original seven hidden layers at 5,000 iterations:



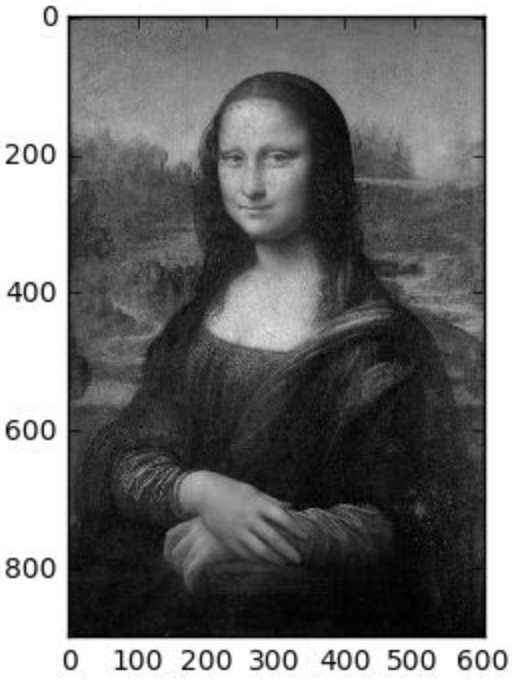
3.4 Random Forests

3.4a. Loading a photo

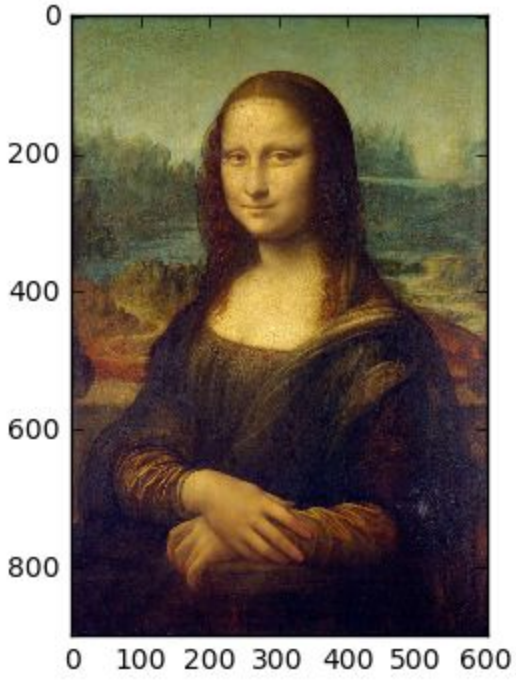
We used Python's PIL (Python Image Library) to represent photos and pixel libraries. The library conveniently allows for image conversion to grayscale, which was helpful in this exercise. We used matplotlib's pyplot library in order to view images created by the pixel arrays we loaded and manipulated.

The image we chose to train our model on was the Mona Lisa image taken from the Wikipedia entry dedicated to the famous painting.

```
im = Image.open("Mona_Lisa.jpg")
im_gray= im.convert('LA')
plt.imshow(im_gray)
plt.show()
```




```
plt.imshow(im)
plt.show()
```



3.4b-c. Preprocessing the input

We sampled 5,000 randomly chosen (x,y) coordinates from the image. We also obtained the brightness levels of these randomly selected pixels, and saved them in a separate list.

```
X=[]
gray_pixels = []
for i in range(5000): #choosing random 5000 pixels
    ran_x=random.random()*1000
    if ran_x>=604:
        ran_x/=10
    ran_x=int(ran_x)
    ran_y=random.random()*1000
    if ran_y>=900:
        ran_y/=10
    ran_y=int(ran_y)
    X.append([ran_x,ran_y]) #contains the 5,000 train coordinates
    gray_rgb, ran = im_gray_load[ran_x,ran_y]
    gray_pixels.append(gray_rgb) #contains the brightness of the coordinates in X
```

The initial image we obtained was made of a random sample of the fully-colored original image. We will show in part 2e that the restructured color image was hard to work with because of the new color palette that emerged. We therefore decided to opt for preprocessing the input by converting the image to grayscale.

3.4d. Rescaling the pixel values

We test and found that the image in question was of size 604x900, i.e. it contains 543,600 pixels in an image that is 604 pixel wide and 900 pixel high. Each of those pixels, in the PIL representation that we chose for this task, is a tuple that contains three values: red, green and blue. Each of these values lies in the range [0,255]. In the grayscale, each pixel contains only one value that changes, which represents brightness of the gray image. In PIL, each (x,y) coordinate of this grayscale image contains two values -- one represents brightness, and another that is constantly set to 255.

`im_gray_load[0,0]`	`im_gray_load[200,200]`	`im_gray_load[400,400]`	`im_gray_load[600,890]`
`(109, 255)`	`(68, 255)`	`(36, 255)`	`(3, 255)`

Initially, we rescaled each pixel so that its values lie in the range [0.0,1.0]. However, as we continued in the exercise, we found that we were only to revert the values back to the range [0,255]. We realized that because we opted to use grayscale, there was not much use in using the rescaled values except for the fact

that it was good practice in general, as many methods (though not random forests) assume a zero mean and unit variance.

```
#2d - standardize
standardized_pix=[]
standardized_elem=0.0
for i in range(len(gray_pixels)): #5000 loop
    standardized_elem = gray_pixels[i]/255.0 #rescaling to [0,1]
    standardized_pix.append(standardized_elem)
```

3.4e. Random Forest and Preprocessing

The data, in the case of this example, is ready for random forest analysis. We have an array of a fixed size (603x900), whose values are on the same scale [0,255]. There is no additional work required in order to prepare the data for a random forest analysis. The random forest method takes subsets of the data randomly, and uses decision trees in order to perform classification or regression on the data. Decision trees require little preprocessing, and so do random forests. Therefore, we did not employ any further preprocessing methods.

3.4f. Final image

In order to build the final image, we used the RandomForestClassifier from ScikitLearn.

We first built a 543,600-cell long list, which contains all of the possible (x,y) coordinates in the original image:

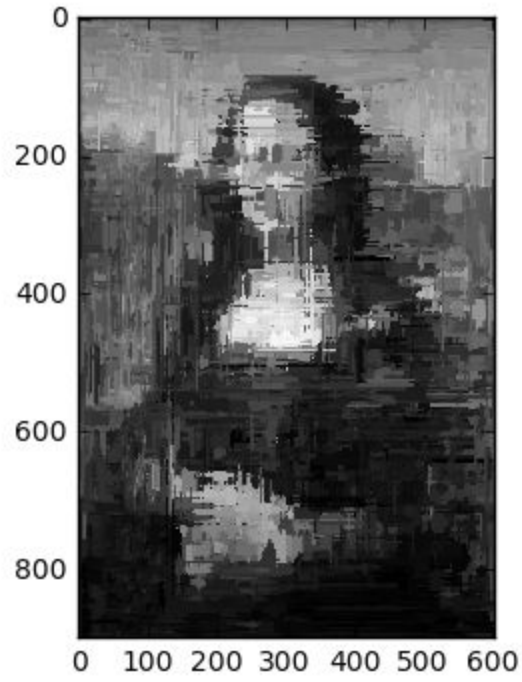
```
In [19]: height,width = im.size
         all_pixels=[]
         for i in range(height):
             for j in range (width):
                 all_pixels.append((i,j))

In [20]: len(all_pixels)

Out[20]: 543600
```

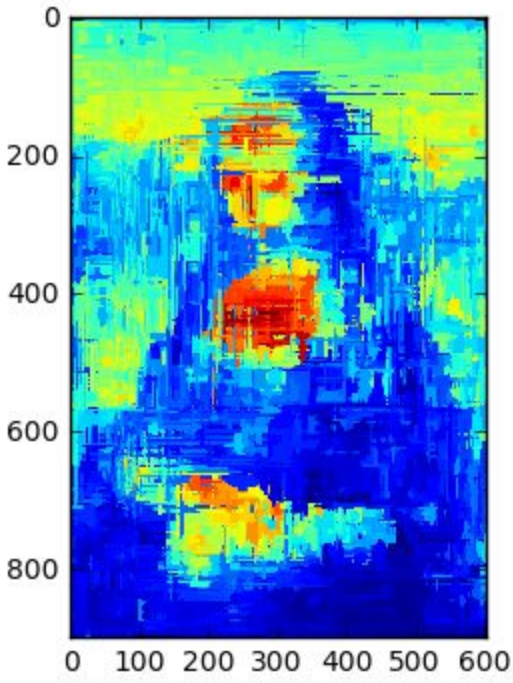
We fit the classifier on the random sample of 5,000 (x,y) coordinates from the original Mona Lisa image. We then restructured the image using the classifier and the list of all coordinates:

```
def experimentation (X_train, num_estimators, depth, image_size_array, pixels):
    forest = RandomForestClassifier(n_estimators=num_estimators, max_depth=depth)
    forest.fit(X_train, pixels)
    prediction = forest.predict(image_size_array)
    plt.imshow(np.array(prediction).reshape(height, width), cmap="gray")
    plt.show()
```

We then tested what the image would look like, when restructured using a random forest of 10 decision trees without a limit to the maximum depth:

```
#2f
experimentation(X, 10, None, size_arr, gray_pixels)
```

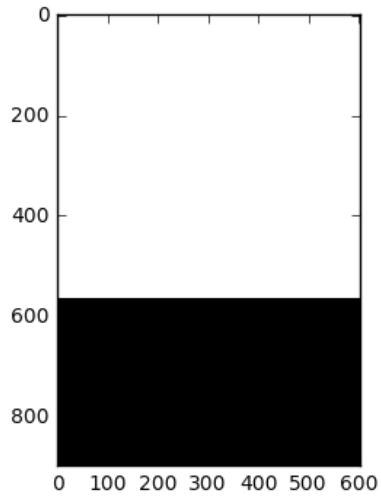




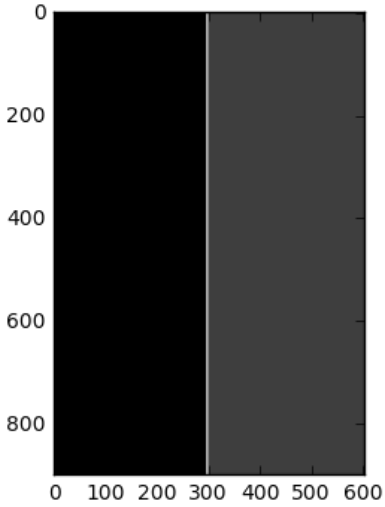
On the righthand side, it is noticeable that the image restructured from the original colored image contains a color scheme that is very different to the original. We preferred to avoid working with this palette, and stuck to the grayscale analysis, which is documented in this report.

3.4g-a. Changing depth with a single decision tree

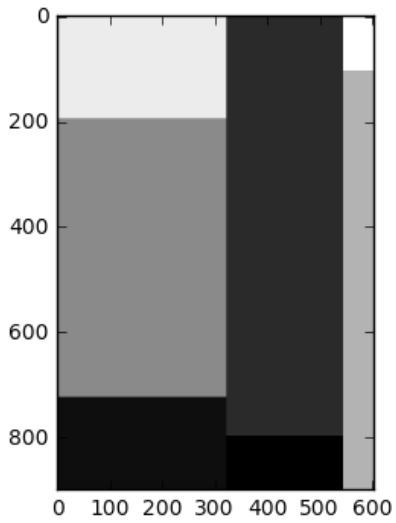
single decision tree with depth as 1



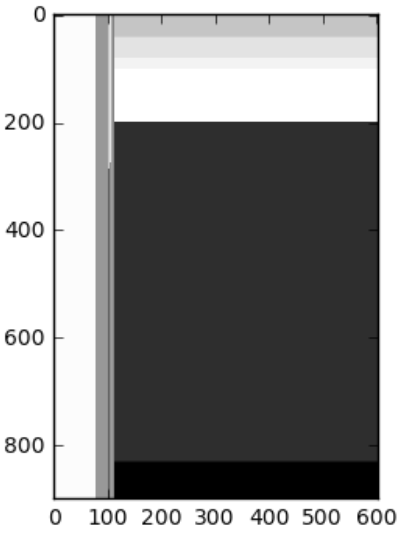
single decision tree with depth as 2



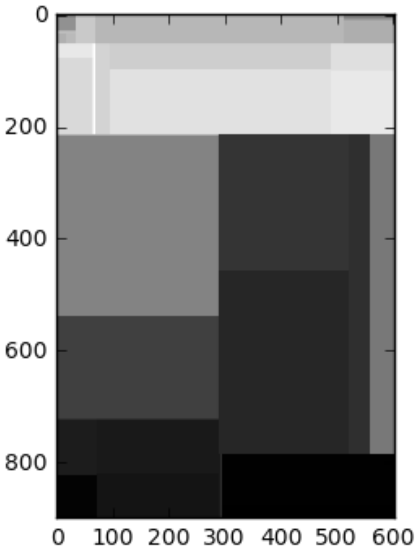
single decision tree with depth as 3



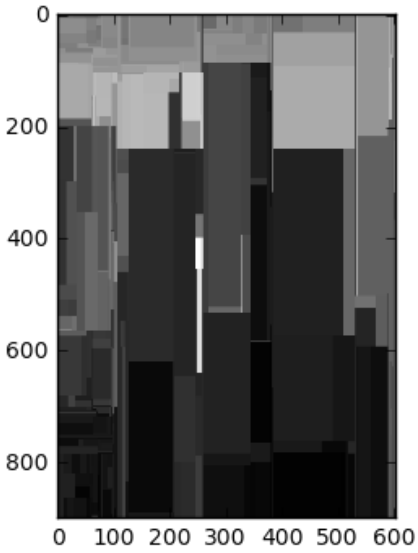
single decision tree with depth as 4



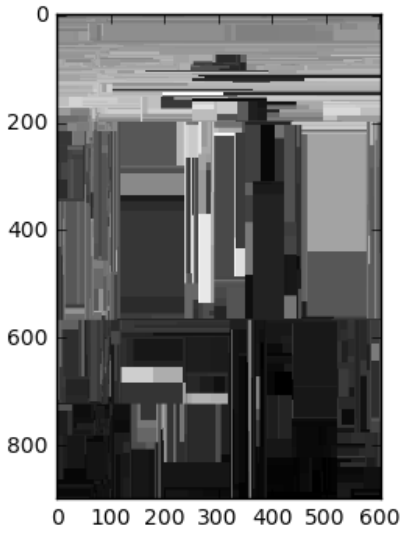
single decision tree with depth as 5



single decision tree with depth as 10



single decision tree with depth as 15



A random forest is a collection of unpruned decision trees. Each decision tree is built from a random subset of the training dataset in performing this sampling. That is, some entities will be included more than once in the sample, and others won't appear at all. Generally, about two thirds of the entities will be included in the subset of the training dataset, and one third will be left out (reference 6).

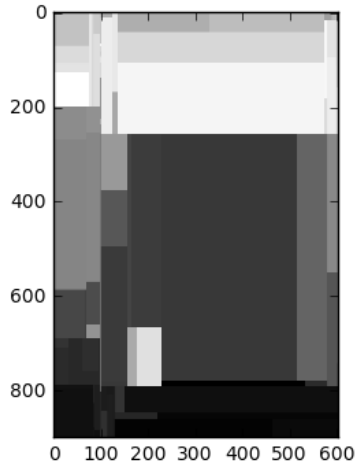
By restricting the random forest to a single decision tree, we are effectively not allowing it to look at different subsets of the data and use different decision trees

to sort them. Moreover, by restricting this single tree to a maximum depth level of 15, we are only letting it sort the data into 15 subgroups, which in this case are 15 shades of gray. We can see that with a depth of 1, the decision tree sorts the image into two groups (two shades: black and white), then the number of groups increases directly with the depth of the image. Eventually, we see that a single decision tree with depth of 15 can start seeing more nuances of color, but still very far away from what is necessary to “see” the Mona Lisa.

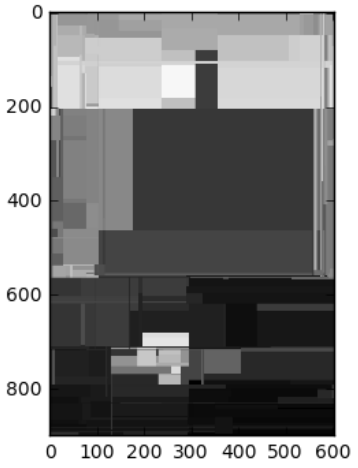
3.4g-b. Changing decision trees with depth of 7

Next, we looked at what happens when we toggle the number of decision trees in the random forest, while maintaining a depth of 7 nodes.

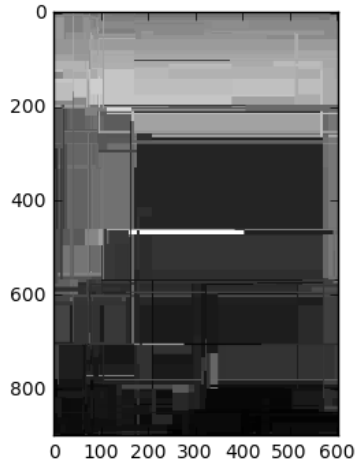
depth is 7 while number of trees is 1



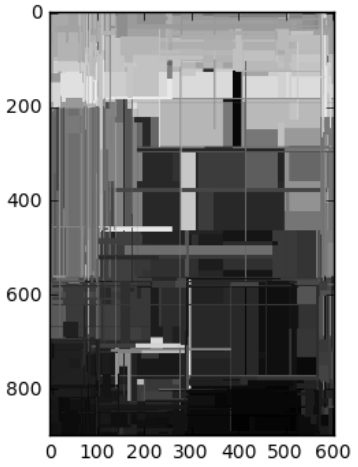
depth is 7 while number of trees is 3



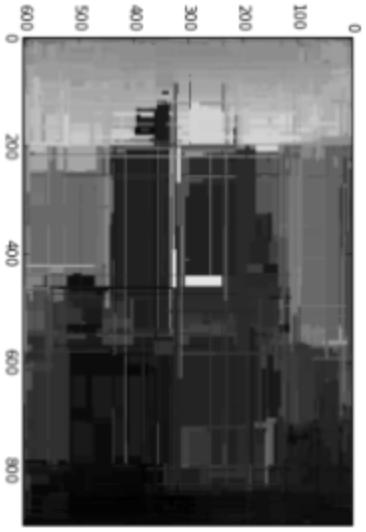
depth is 7 while number of trees is 5



depth is 7 while number of trees is 10



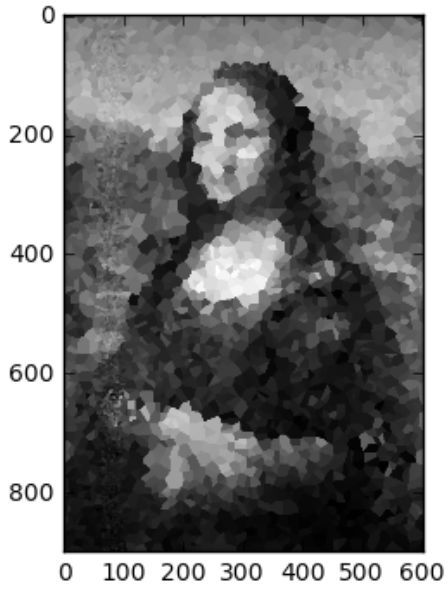
Despite our best efforts, we could not get any computer to run a random forest with 100 trees or even 50 trees without the kernel crashing. We managed to run a forest algorithm with 25 trees:



3.4g-c. KNN

We ran a K-nearest-neighbors algorithm (K=1) on the training set. As is visible in the image below, the algorithm yielded an image divided into small angular blocks of color. In K-nearest-neighbors, each coordinate “queries” the class of other pixels closest to it, and the plurality vote determines the class of that pixel. In K=1, each pixel gets its class (in our case, its brightness) from the nearest pixel to it. In places where the original image had a delicate interplay between light and shadow, the KNN-constructed image has some darker shapes interlaced with lighter ones, but their boundaries are clear; there is no blend of colors. In areas where there is an interplay of light - most notably on the Mona Lisa’s chest and her right arm - there are much lighter shapes placed next to darker ones. Mona Lisa’s face and chest are the fairest parts of the image, and they contain some nearly white-bright shapes in the KNN-constructed image, which indicates that their class was determined by some of the most brightly lit parts of the image.

```
#2g-c
from sklearn.neighbors import KNeighborsRegressor
knn = KNeighborsRegressor(n_neighbors=1)
knn.fit(X, gray_pixels)
knn_predict = knn.predict(size_arr)
to_view = np.asarray(knn_predict)
to_view = to_view.reshape(height,width)
plt.imshow (to_view, cmap = "gray")
plt.show()
```



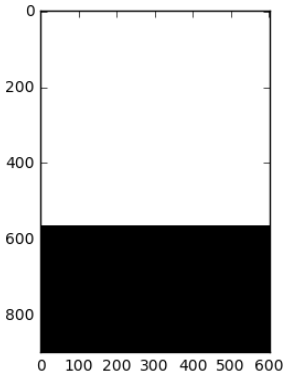
3.4g-d. Pruning strategies

One of the common issues with random forests and, in general, with decision trees is that a small tree might not capture important structural information about the sample space, and a large one might overfit the data. Pruning allows us to grow the tree until each node contains enough number of instances to determine its likely class, and remove nodes that don't provide additional information.

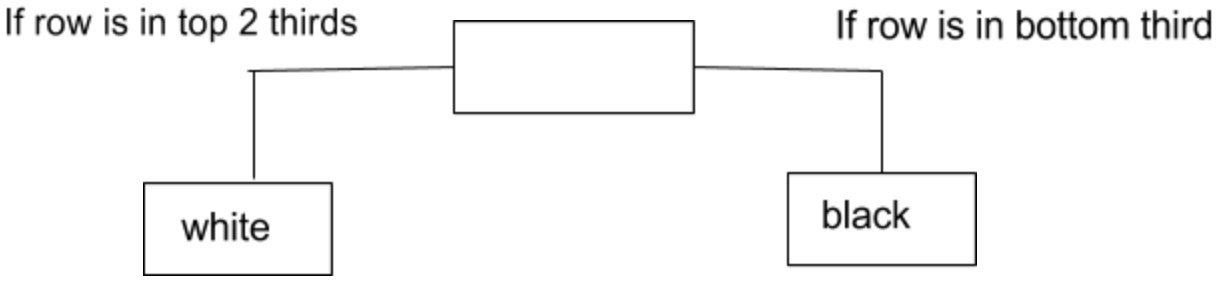
In particular, we've looked into several implementations of reduced error pruning (see reference 3). We noticed that, in order to conduct pruning, there is a need of knowing in advance which classes we are expecting to see in the decision tree. In our case, we would be looking for different levels of brightness or different shades of gray. However, because we trained our random forest on 5,000 random pixel from the 543,600-pixels image, it appeared to us that pruning would not be necessary: . As the implementation of pruning seemed more costly than the method we chose

2h-a. Decision rule

Let's look at the following image, structured by a single decision tree with depth 1:

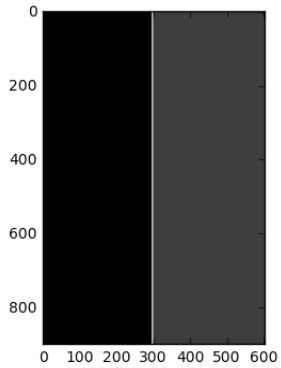


This image was structured by a single tree with a single split:

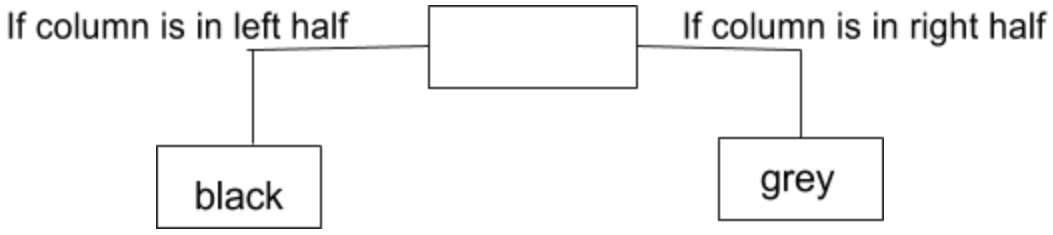


Or, in a formula:
total_rows, width = image.size
If row <= total_rows*⅔:
    return white
else:
    return black

Let's look at the following image, structured by a single decision tree with depth 2:



We can see here a similar breakdown, but this time on the vertical axis of the image instead of the horizontal one:

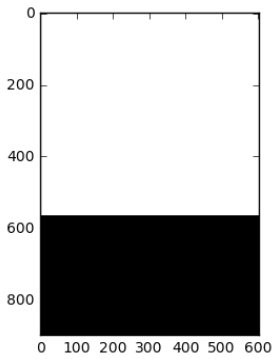


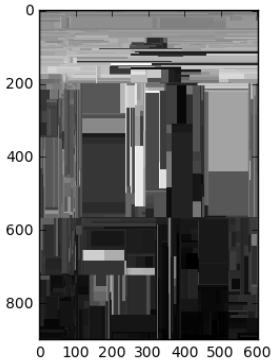
Looking at the original image, it can be inferred that these initial formulae correspond with the overall breakdown of image brightness; i.e. the original image is lighter in the top two-thirds, and its left half is slightly darker than its right half.

3.4h-b. Decision rule

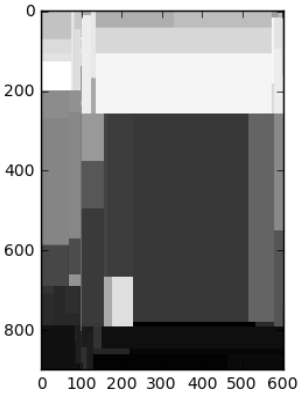
Let's compare two sets of trees that we had built earlier:

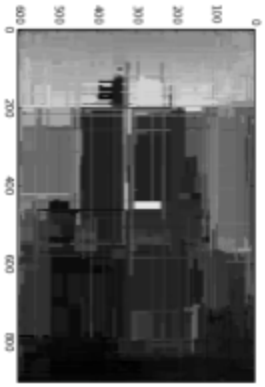
Both forests below are made of a single decision tree; the one on the left has depth of 1, the one on the right has depth of 15.





Both forests below contain trees that have depth 7; the one of the left has a single decision tree, the one on the right has 25 trees.





As the images above show us, both the number of decision trees in the forest and their depth have an effect on the patches from which the image is composed. The larger the number of decision trees that make up the forest, the more nuance there is in brightness: the boundaries between the different patches become less sharp, and there is a greater number of patches that are in different shades of gray. In addition, the depth of each tree -- i.e. the number of intersections/nodes at which data is being filtered -- affects the patches in the following manner: as the depth of the tree(s) in the forest increases, the number of patches in the reflected image increases, and at the same time the size of each patch decreases. As the patches become smaller, they each get assigned a more nuanced shade of gray; a tree with depth of 100 has more patches of smaller size, and in more shades than a tree with depth of 10. This quality, of course, affects the image conveyed by the forest that contains this tree and others like it. The brightness level (i.e. the shade of gray) of a particular patch is determined by the other pixels in its vicinity.

3.4h-c. Patches equation - easy.

We defined as the depth of the decision tree.

Number of patches of color = 2^d

3.4h-d. Patches equation - tricky.

If a single tree yields 2^d patches of color, then n decision trees could yield n(2^d) patches of color if every single tree in the forest is different from the other. In the very unlikely case that every single tree in the forest is identical, then we would have an image with 2^d patches of color. And so, the answer is that we would see between 2^d and n(2^d) patches of color.

4. References

1. http://neuralnetworksanddeeplearning.com/chap4.html
2. https://en.wikipedia.org/wiki/Association_rule_learning
3. https://en.wikipedia.org/wiki/Artificial_neural_network
4. https://en.wikipedia.org/wiki/Random_forest
5. http://cs.stanford.edu/people/karpathy/convnetjs/demo/image_regression.html
6. http://datamining.togaware.com/survivor/Random_Forests.html
7. https://en.wikipedia.org/wiki/Pruning_(decision_trees)
8. https://triangleinequality.wordpress.com/2013/09/01/decision-trees-part-3-pruning-your-tree/