CHANDRIMA BHATTACHARYA

&

ZEYU WANG

Applied Machine Laerning

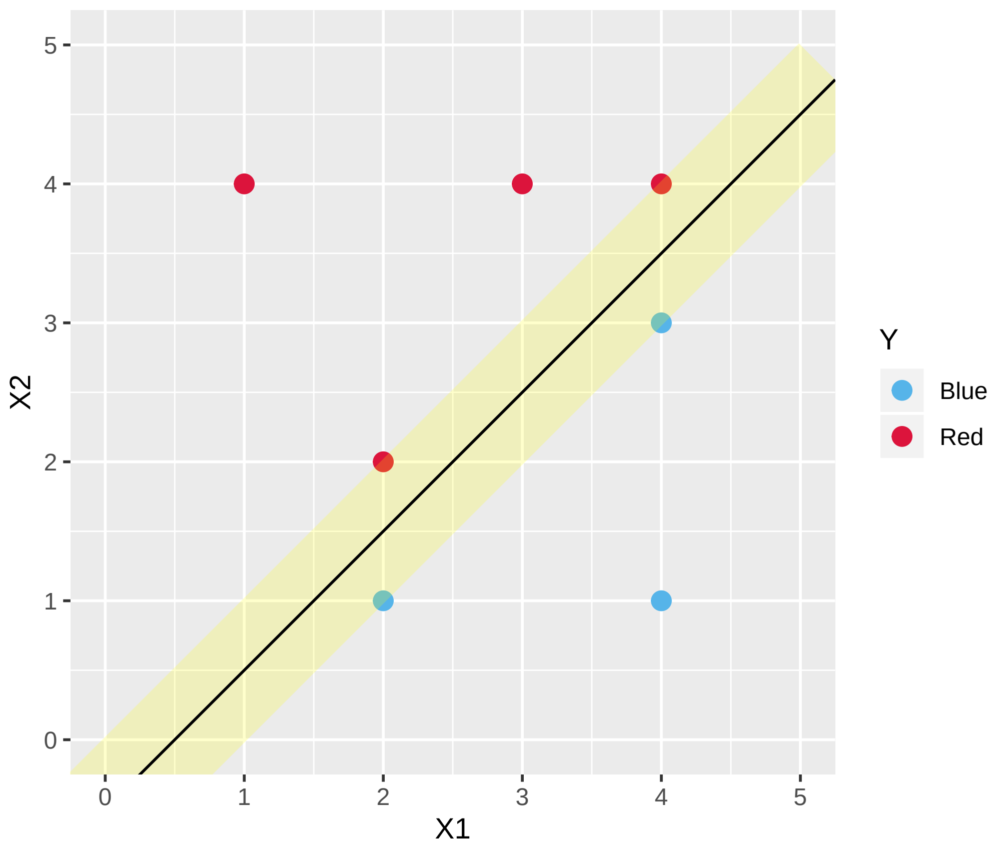
Homework 4 Written Part

Zeyu Wang, Chandrima Bhattacharya

12/2/2019

Question 1

Graph for 1.a ~ 1.d



1.a

As graph shows above, points have been shown in color based on their value in "Y", the black line shows the maximum-margin classifier.

1.b

The classification rule of this classifier should be:

$$Y = \begin{cases} Red, & \text{if } X_1 - X_2 - 0.5 < 0 \\ Blue, & \text{if } X_1 - X_2 - 0.5 > 0 \end{cases} \beta_0 = -0.5, \quad \beta_1 = 1, \quad and \quad \beta_2 = -1$$

1.c

The margin shows in light yellow fields around the hyperplane.

1.d

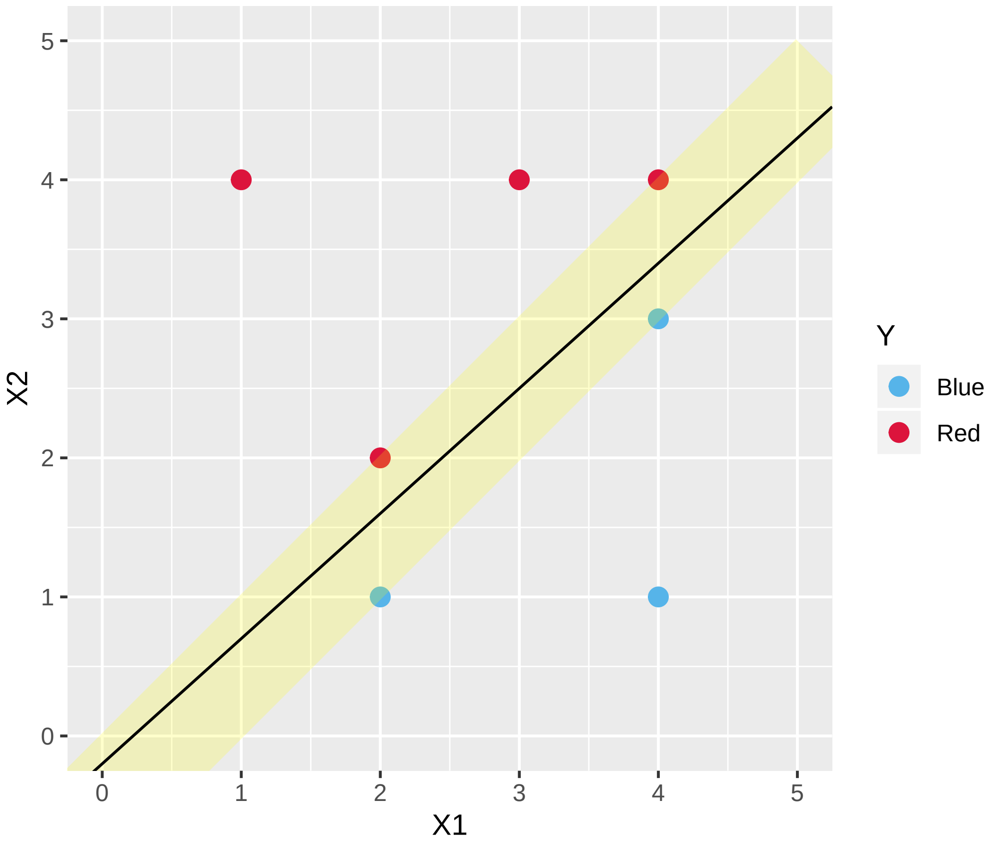
The support vectors are four points that on the boundary of margin, which are (2,1), (4,3), (2,2) and (4,4).

1.e

Since the hyperplane is decided mainly on the margin, which is based on support vectors, a slight movement of the seventh observation (4,1) would not really have effect on the hyperplane, unless it moves inside margin by any chance.

1.f

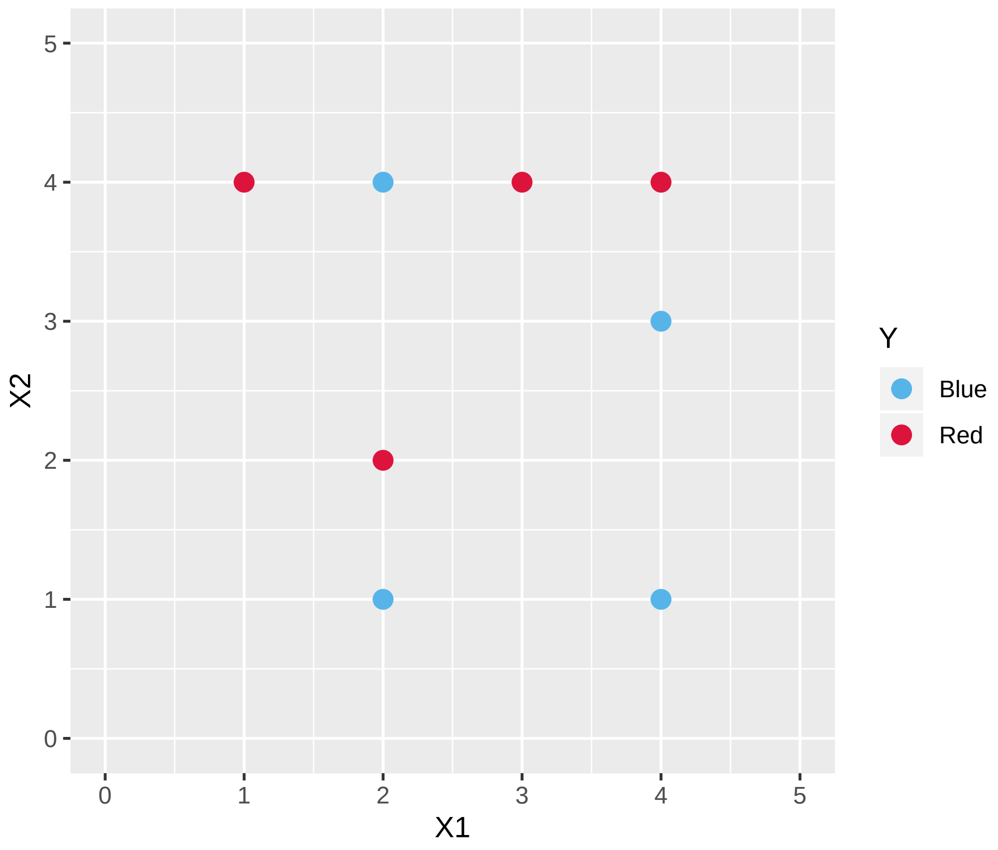
The hyperplane other than maximum-margin hyperplane shows below:



And the equation of this hyperplane is $0.9X_1 - X_2 - 0.2 = 0$**.**

1.g

The additional observation I add is (2,4,“Blue”):



Question 2

As problem shows, we have the input $X \in [0, 10]$ and output is:

$$Y = \begin{cases} 2(X-1), & X \in [1,2) \\ \frac{1}{3}(X+4), & X \in [2,5) \\ 2(X-\frac{7}{2}), & X \in [5,6) \\ -\frac{5}{3}(X-9), & X \in [6,9) \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Based on equation (1), we can then get the output format in one equation:

$$Y = \sigma(2\sigma(X-1) - \frac{5}{3}\sigma(X-2) + \frac{5}{3}\sigma(X-5) - \frac{11}{3}\sigma(X-6)) \tag{2}$$

where $X \in [0, 10]$, σ means ReLU nonlinearity function.

Based on the equation (2), we can find that it is a combination of four ReLU nonlinearity functions, so we can design our neural network as:

- One hidden layer Y_1 with four units
- Connection is full, which means it is a vanilla network
- Functions of units are $\sigma(X-1)$, $\sigma(X-2)$, $\sigma(X-5)$, $\sigma(X-6)$
- Which means the weight of this layer is $W_1 = [1,1,1,1]$ while the bias is $[-1,-2,-5,-6]$
- The output of this layer is $Y_2 = \sigma(W_2 Y_1^T + \beta_2)$, where $W_2 = [2, -\frac{5}{3}, \frac{5}{3}, -\frac{11}{3}]$ and $\beta_2 = \vec{0}$

As a result, for each input value of X, the whole process would be:

1. Put X value as Y_0 to all units of the hidden layer, calculate results of all units;

2. Get the output vector as Y_1, then calculate output value Y_2

3. For $X \in [1,9]$, the output value is the same as what equation (1) shows

4. For $X < 1$ or $X > 9$, because of the ReLU nonlinearity of output function, their negative output would become 0.

Question 1

i. There are 9 layers in this network. The first layer is the input layer. Since we have 2-D data out input is in x and y. The 7 layers in between are FC layers containing 20 neurons each. They are the hidden layers of the neural network. These layers detect specific global configurations of the features detected by the previous layers in the net. Each node in the FC layer learns its own set of weights on all of the nodes in the layer below it. The activation functions used in this structure is a Rectified Linear unit. The last layer does regression on the data fed by previous layers.

ii. The "loss" here is the weighted sum of the current lossand the loss from the lass iteration.

iii. Plot the loss over time, after letting it run for 5,000 iterations.

In [1]:

```
import os
from matplotlib.pyplot import plot
os.chdir(r"F:\Annie\CornellMS\Semester 4\Machine Learning\Homework\HW4\NeuralNets")
```

In [2]:

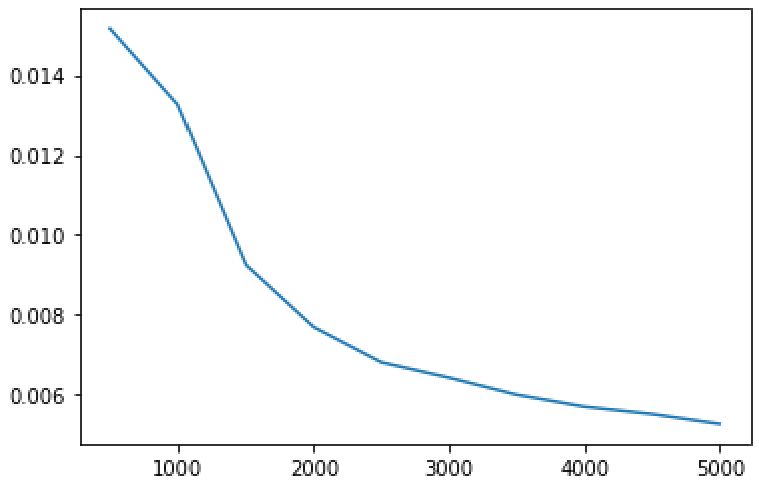
```
loss = [0.015169390486953024, 0.0132662342637763, 0.009249658057037556, 0.00769338172049829
iterations = [499, 999, 1499, 1999, 2499, 2999, 3499, 3999, 4499, 4999]
```

In [3]:

```
plot(iterations,loss)
```

Out[3]:

```
[<matplotlib.lines.Line2D at 0x2555376c860>]
```



The network as we can see from the graph eventually gets better over time. The loss reduces consuderably after 500 iterations. There is a gradual drop in loss after that. And with every iteration loss keeps reducing a bit.

iv. Lower learning rate every thousand

In [4]:

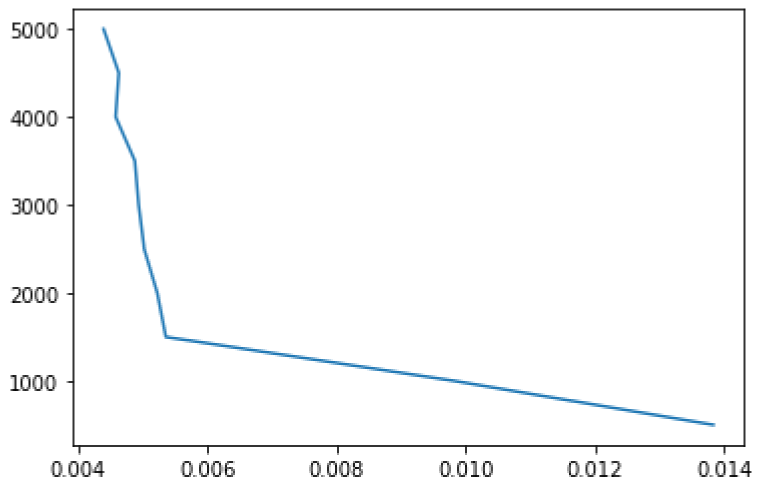
```
loss = [0.013828413061813483, 0.009819505938392875, 0.005355846121161638, 0.00522338224928
iterations = [499, 999, 1499, 1999, 2499, 2999, 3499, 3999, 4499, 4999]
```

In [5]:

```
plot(loss, iterations)
```

Out[5]:

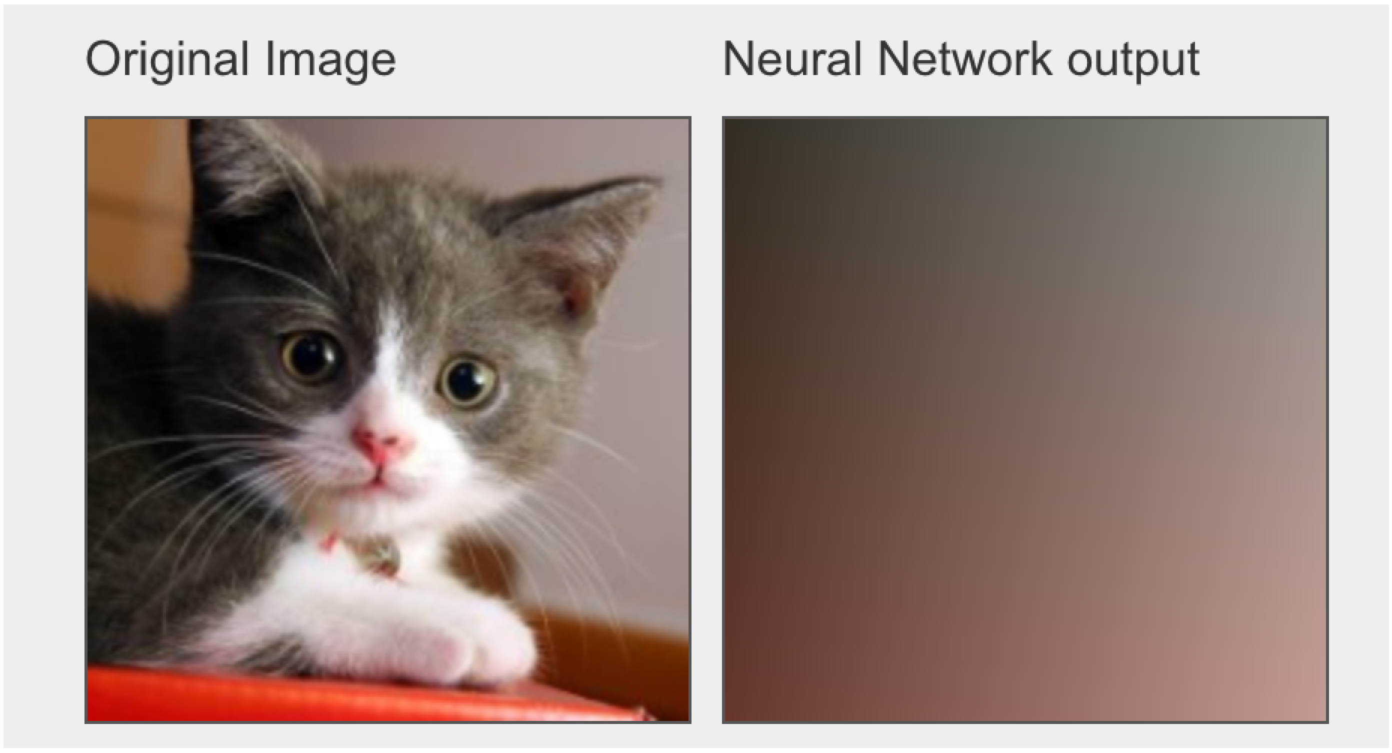
```
[<matplotlib.lines.Line2D at 0x255538117f0>]
```



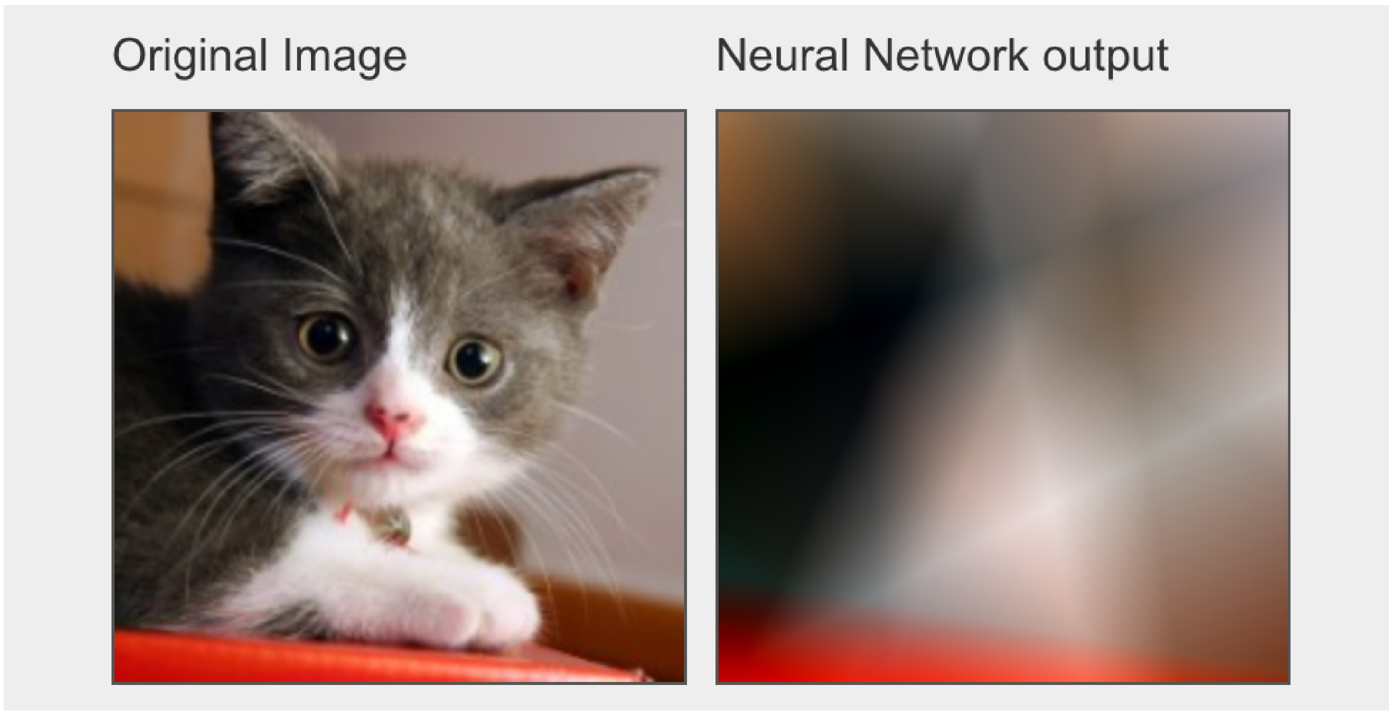
Because of the reduction in learning rate over time, there is not much improvement in the loss function. But there is a steeper decline of the loss function.

v. Lesion Study

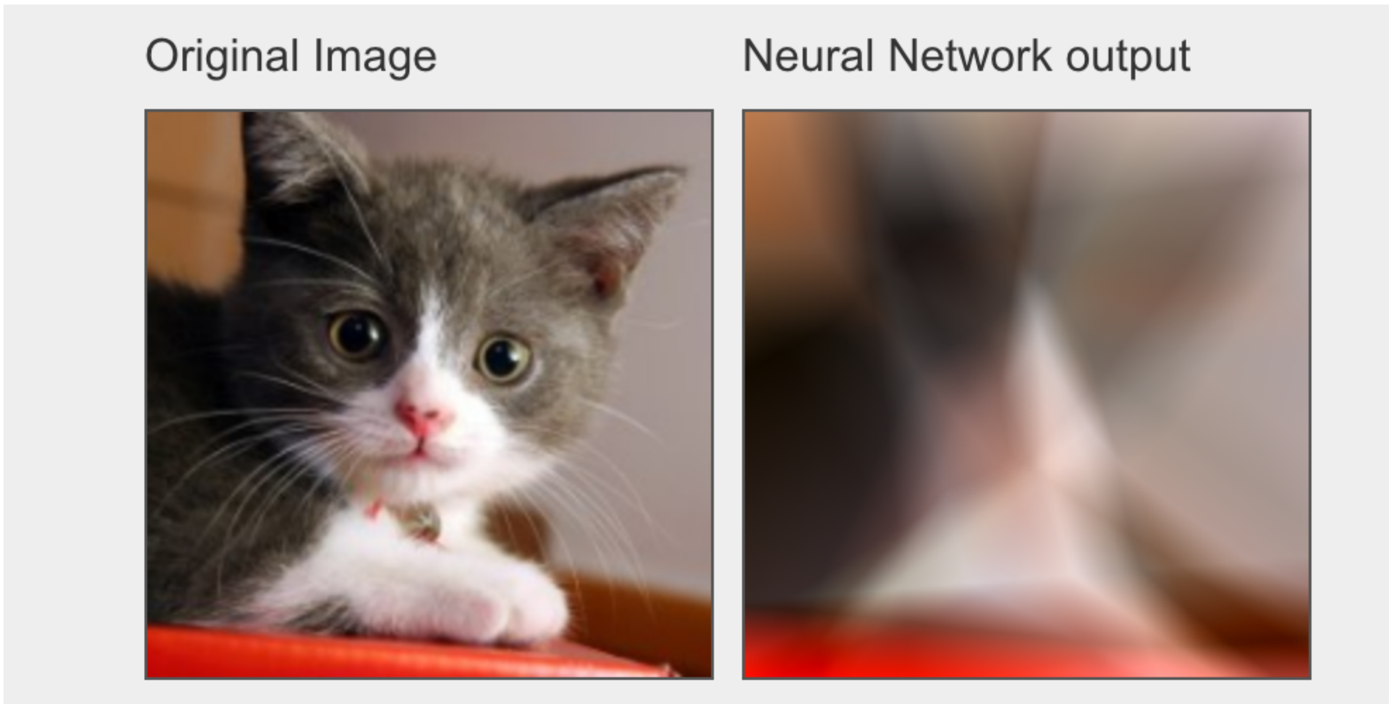
```
0 hidden layers
Dropping all the hidden layers, keeping only the regression layer.
After 1000 iterations the loss is 0.07
The Neural Network Output barely resembles the original image.
```



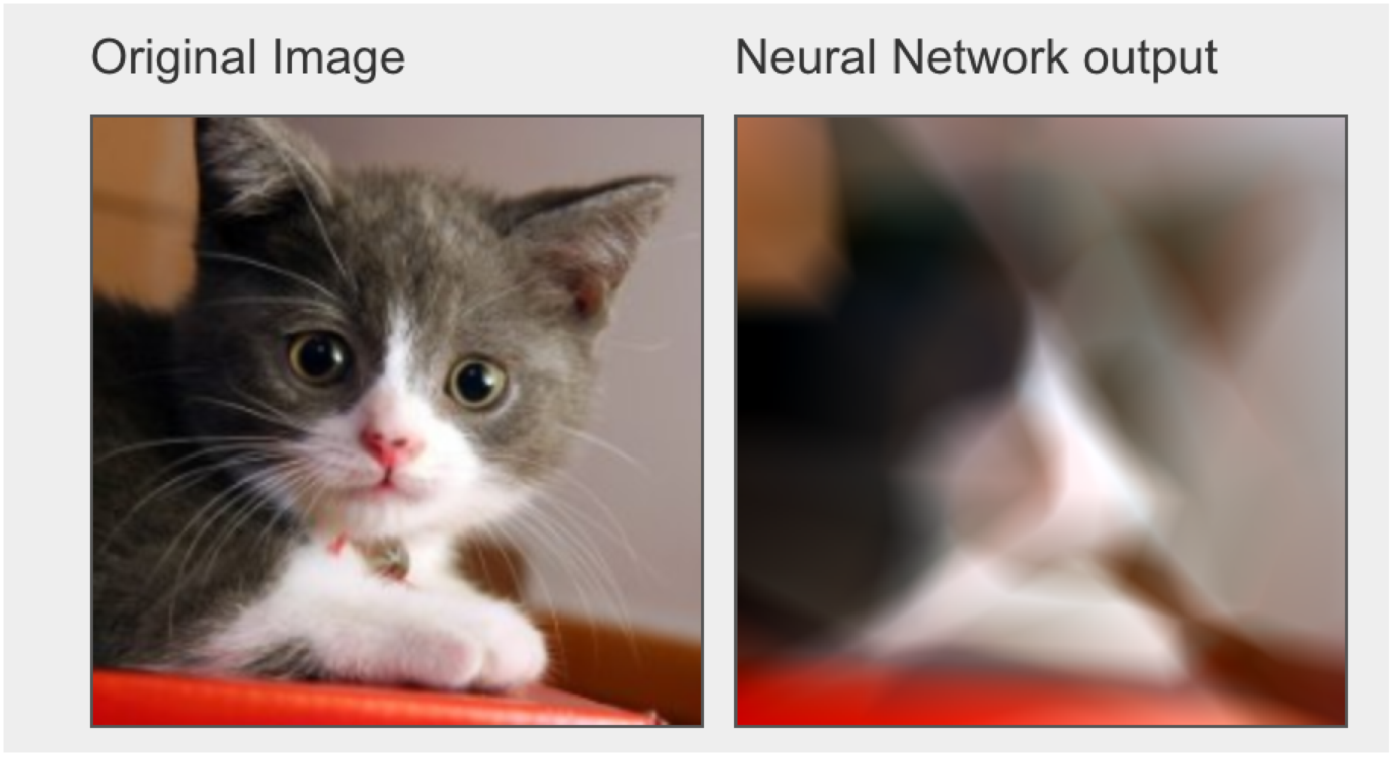
1 hidden layer with 20 neurons The loss reduces from 0.07 to 0.03 with the introduction of one hidden layer with 20 neurons



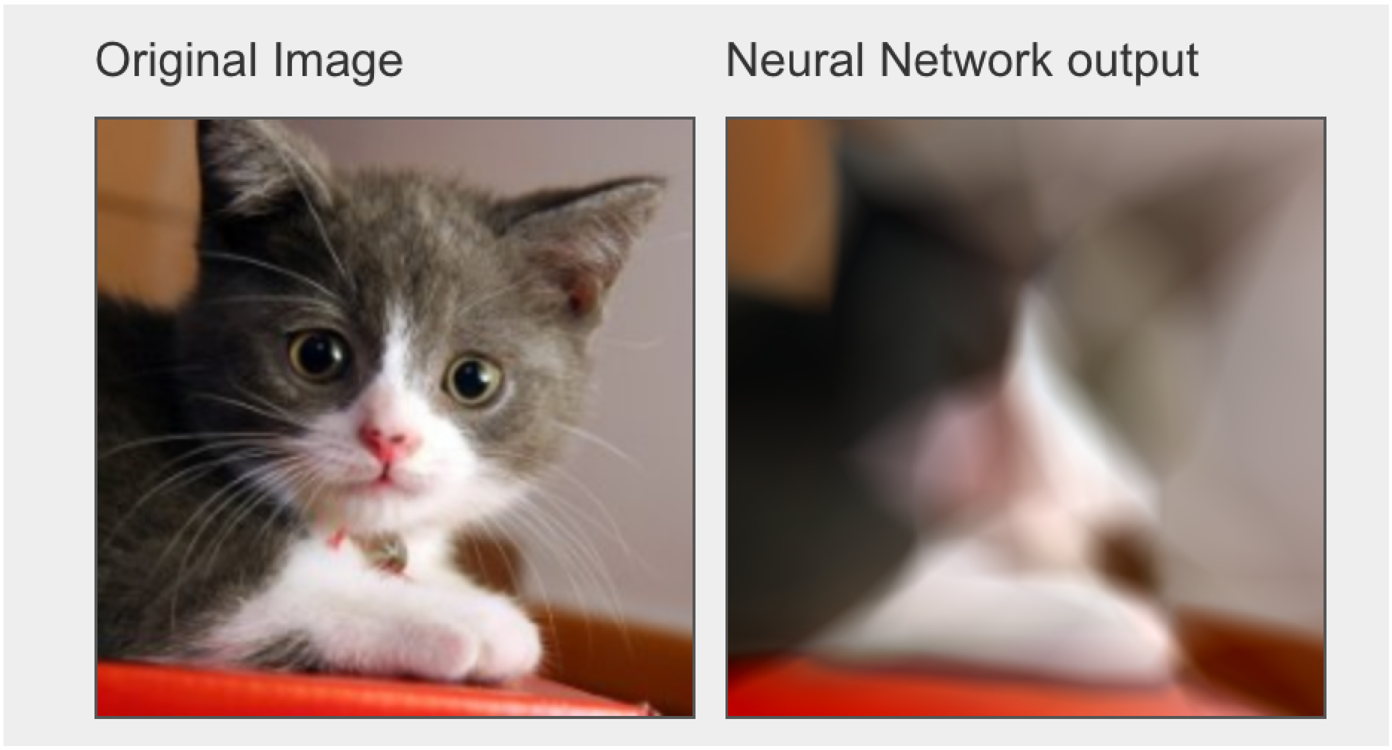
2 hidden layers with 20 neurons each Takes much longer time to run with 2 layers. Loss drops to 0.02 in the first 400 iteration itself. The whites and the blacks are more prominent. After a 1000 iterations the loss comes to 0.018



3 hidden layers with 20 neurons each The loss comes to 0.01 after a 1000 iterations. The network image gives a much better idea of the cat.



4 hidden layers The loss has come down to 0.011 after 1000 iterations.



2 hidden layers are good enough.

f: Adding more layers

A total of 9 layers brought the loss down to 0.008

```
In [ ]:
```

Question 2

In [1]:

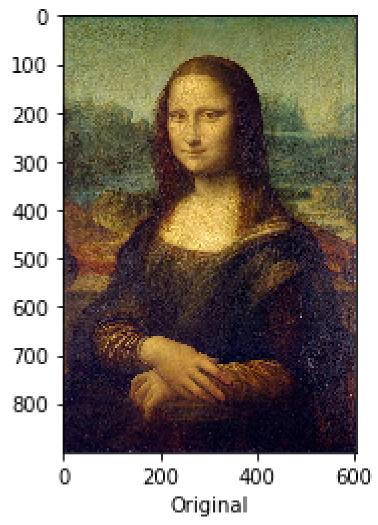
```
import numpy as np
import matplotlib.pyplot as plt
import os
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor
import warnings
warnings.filterwarnings("ignore")
```

In [2]:

```
# a: import picture
pic = plt.imread('F:\Annie\CornellMS\Semester 4\Machine Learning\Homework\HW4\Random_Forest
plt.imshow(pic)
plt.xlabel('Original')
```

Out[2]:

```
Text(0.5, 0, 'Original')
```



In [3]:

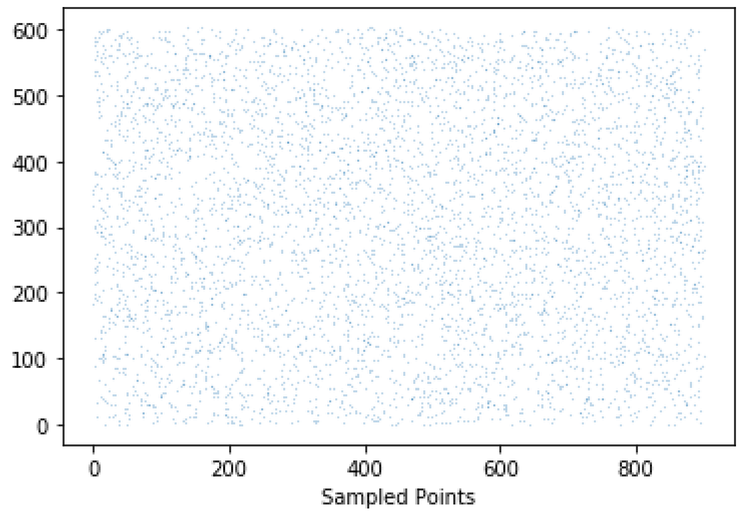
```
# b: Preprocessing input
samples = np.arange(900*604)
samples_arr = samples.reshape(900,604)
random_samples = np.random.choice(samples,5000)

def grab_xy(samples):
    x,y = [],[]
    for i in samples:
        x.append(np.where(samples_arr == i)[0][0])
        y.append(np.where(samples_arr == i)[1][0])
    xy = np.column_stack((np.array(x,dtype=np.int), np.array(y,dtype=np.int)))
    return(xy)

x_tr = grab_xy(random_samples)
print (x_tr.shape)
# create a white image
plt.scatter(*zip(*x_tr), s=0.01)
plt.xlabel("Sampled Points")
plt.show()

row,col = np.indices((900,604))
x_te = np.column_stack((row.reshape(-1,1),col.reshape(-1,1)))
```

```
(5000, 2)
```



We did not use any preprocessing as we are using decision tree. Inparticular,we do not need to perform mean subtraction, standardization, or unit-normalization, as decision tree does not get affected by the following.

In [4]:

```
# c: Preprocess output
def RGB(xy):
    a = []
    for i in range(len(xy)):
        a.append(pic[xy[i][0],xy[i][1]])
    b = np.row_stack(a)
    b = b/255
    return(b)

y_tr = RGB(x_tr)
y_te = RGB(x_te)
```

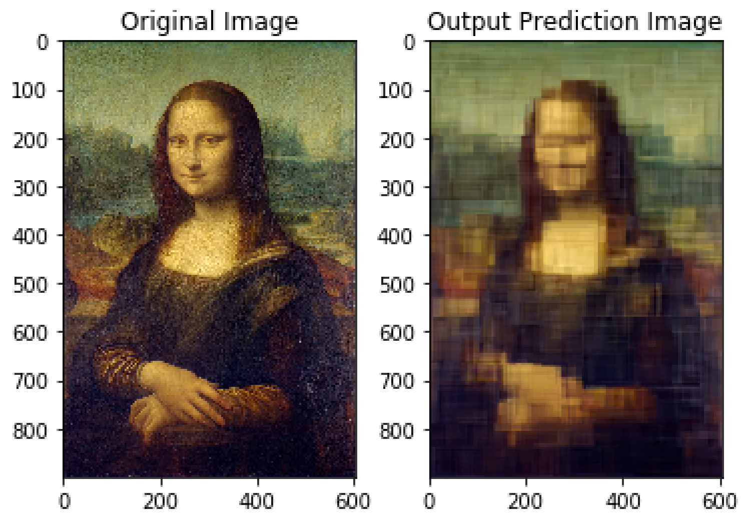
For preprocessing the output, we are using the 2nd process. We are regress all three values at once, so your function maps (x,y) coordinates to (r,g,b) value: f: R2 -> R3

In [5]:

```
# d: Display random forest
clf = RandomForestRegressor()
clf.fit(x_tr, y_tr)
plt.subplot(1,2,1)
plt.imshow(pic)
plt.title('Original Image')
plt.subplot(1,2,2)
pred_pic = clf.predict(x_te).reshape(pic.shape)
plt.imshow(pred_pic)
plt.title('Output Prediction Image')
```

Out[5]:

```
Text(0.5, 1.0, 'Output Prediction Image')
```

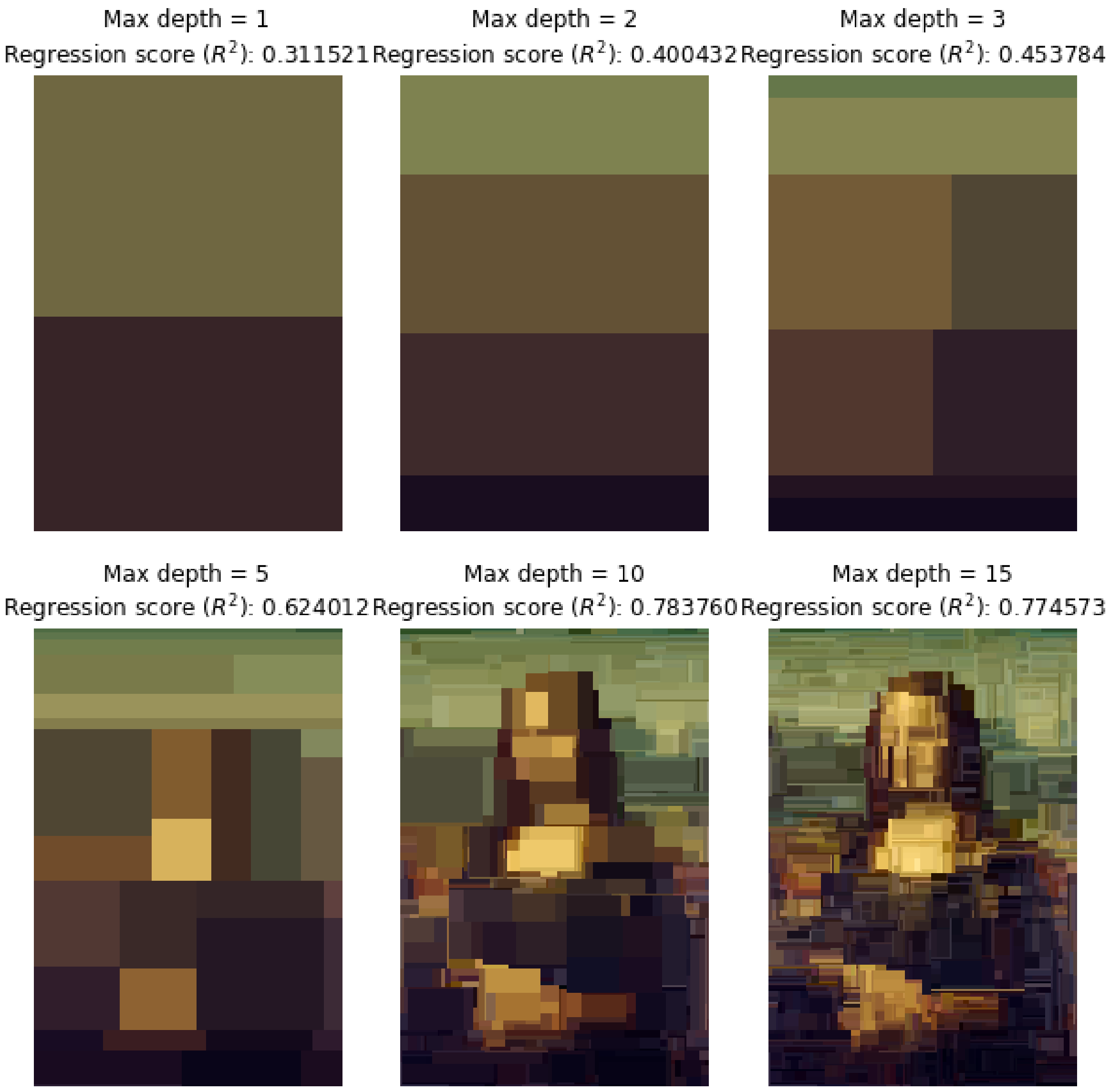


No other preprocessing is needed for using random forest.

In [6]:

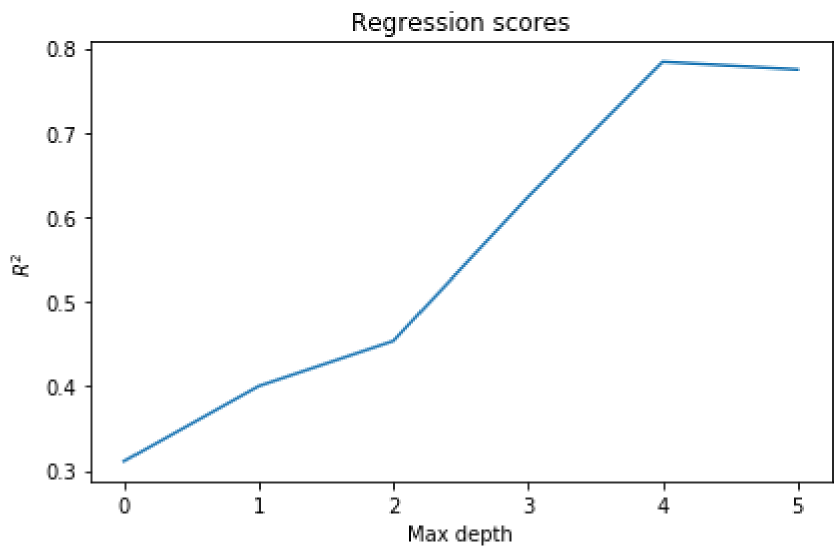
```
# e: Experimentation with random forest
# i: Single decision tree, but with depths 1, 2, 3, 5, 10, and 15

regression_scores = []
depth = [1,2,3,5,10,15]
plt.figure(figsize=[10, 10])
for i, d in enumerate(depth):
    reg = RandomForestRegressor(n_estimators=1, max_depth=d)
    reg.fit(x_tr, y_tr)
    pic = reg.predict(x_te).reshape(pic.shape)
    plt.subplot(2, 3, i + 1)
    plt.imshow(pic)
    plt.axis('off')
    regression = reg.score(x_te, y_te)
    regression_scores.append(regression)
    plt.title('Max depth = %d\nRegression score ($R^2$): %f' % (d, regression))
```


Max depth = 1
Regression score ($R^2$): 0.311521
Max depth = 2
Regression score ($R^2$): 0.400432
Max depth = 3
Regression score ($R^2$): 0.453784
Max depth = 5
Regression score ($R^2$): 0.624012
Max depth = 10
Regression score ($R^2$): 0.783760
Max depth = 15
Regression score ($R^2$): 0.774573

In [7]:

```
plt.plot(regression_scores)
plt.title('Regression scores')
plt.ylabel('$R^2$')
plt.xlabel('Max depth')
plt.tight_layout()
```

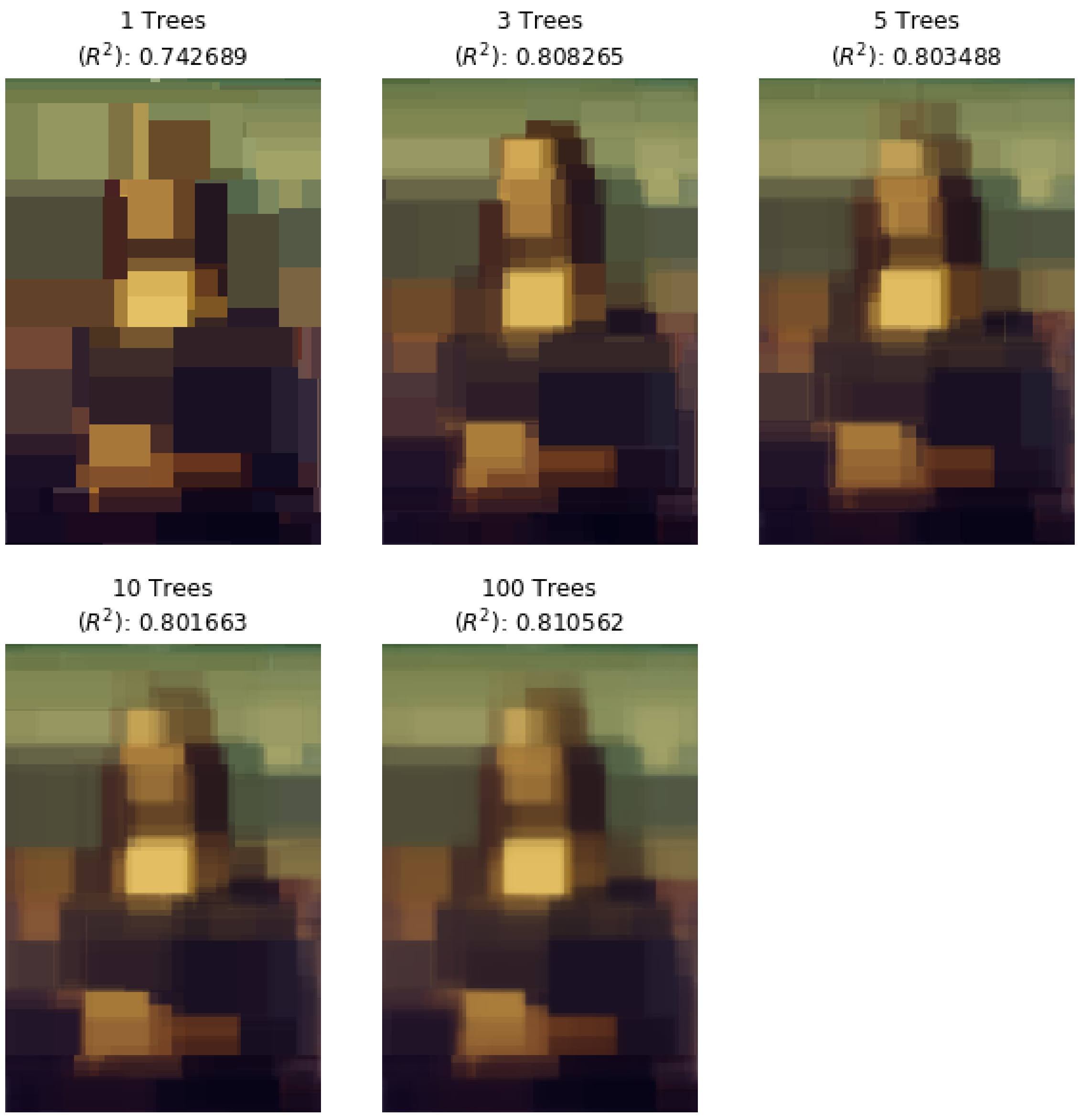


We see that as we keep on increasing the depth, the method understand more diversity of color. For case depth=1, it becomes a binary case of understanding colors.

In [8]:

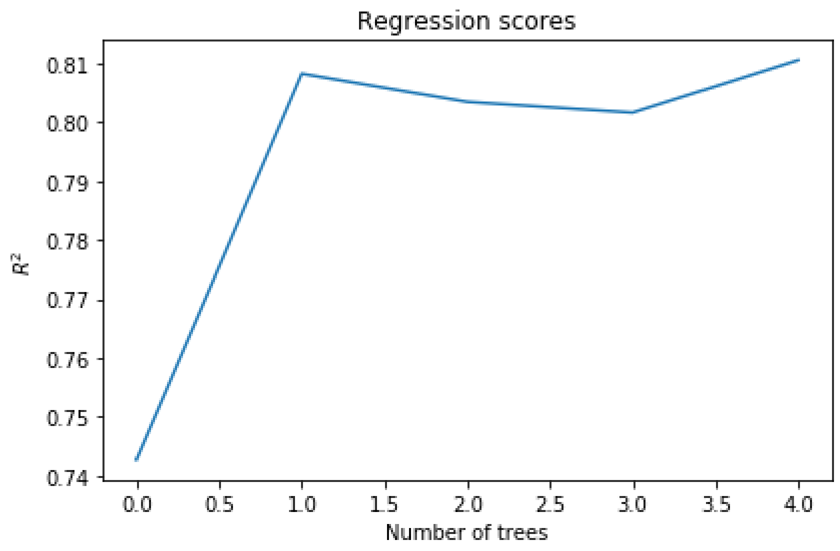
```
# ii: Depth 7, but with number of trees equal to 1, 3, 5, 10, and 100
regression_scores = []
n_trees = [1,3,5,10,100]
plt.figure(figsize=[10, 10])
for i, n in enumerate(n_trees):
    reg = RandomForestRegressor(n_estimators=n, max_depth=7)
    reg.fit(x_tr, y_tr)

    pic = reg.predict(x_te).reshape(pic.shape)
    plt.subplot(2, 3, i + 1)
    plt.imshow(pic)
    plt.axis('off')
    regression = reg.score(x_te, y_te)
    regression_scores.append(regression)
    plt.title('%d Trees\n($R^2$): %f' % (n, regression))
```



In [9]:

```
plt.plot(regression_scores)
plt.title('Regression scores')
plt.ylabel('$R^2$')
plt.xlabel('Number of trees')
plt.tight_layout()
```

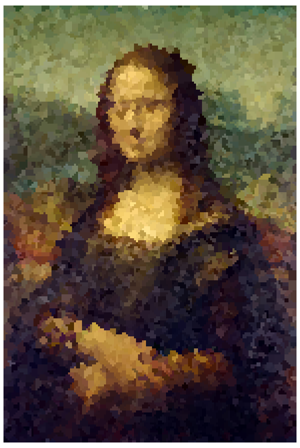


We see that each decision tree will end up yielding a different subset of data, which will be resulting in highly correlated predictor, limiting the variance. Thus by increasing the number of trees, we get smoother image.

In [10]:

```
# c: knn implementation
knn = KNeighborsRegressor(n_neighbors=1)
knn.fit(x_tr, y_tr)
pic = knn.predict(x_te).reshape(pic.shape)
sc = knn.score(x_te, y_te)
plt.imshow(pic)
plt.axis('off')
plt.title('1-NN Regressor baseline\n$R^2=$%f' % sc)
plt.show()
knn.score(x_te, y_te)
```

1-NN Regressor baseline
$R^2 = 0.818786$



Out[10]:

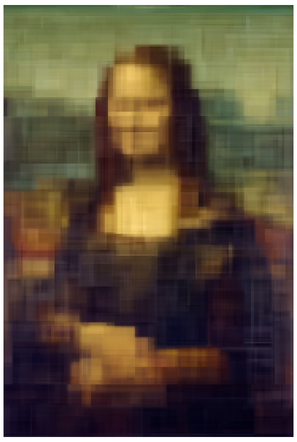
```
0.8187861346688016
```

Here in random forest,we split the image based on pixel position of the image compared with the threshold of each subtree in the forest. This is why the image is divided into small blocks of each colors. However in knn, we are assigning the color to nearest one. Hence we don't have a hard boundary.

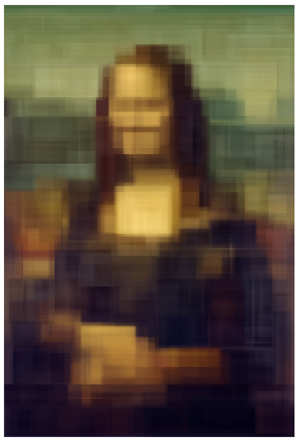
In [11]:

```
# d: Prunning by min_samples_split
n_trees = 50
max_depth = 20
split_list = [5, 10, 20]
regression_scores = []
plt.figure(figsize=[14, 5])
for i, split in enumerate(split_list):
    reg = RandomForestRegressor(n_estimators=n_trees, max_depth=max_depth, min_samples_spli
    reg.fit(x_tr, y_tr)
    pred = reg.predict(x_te).reshape(pic.shape)
    plt.subplot(1, 3, i+1)
    plt.imshow(pred)
    plt.axis('off')
    sc = reg.score(x_te, y_te)
    regression_scores.append(sc)
    plt.title('Min sample split = %d\n$R^2$ = %f' % (split, sc))
```

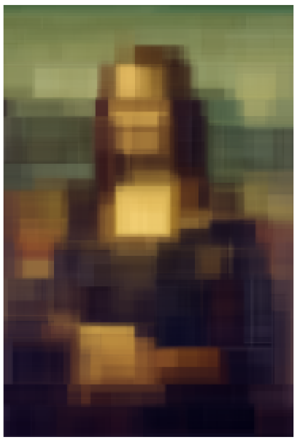
Min sample split = 5
$R^2$ = 0.876274



Min sample split = 10
$R^2$ = 0.868814

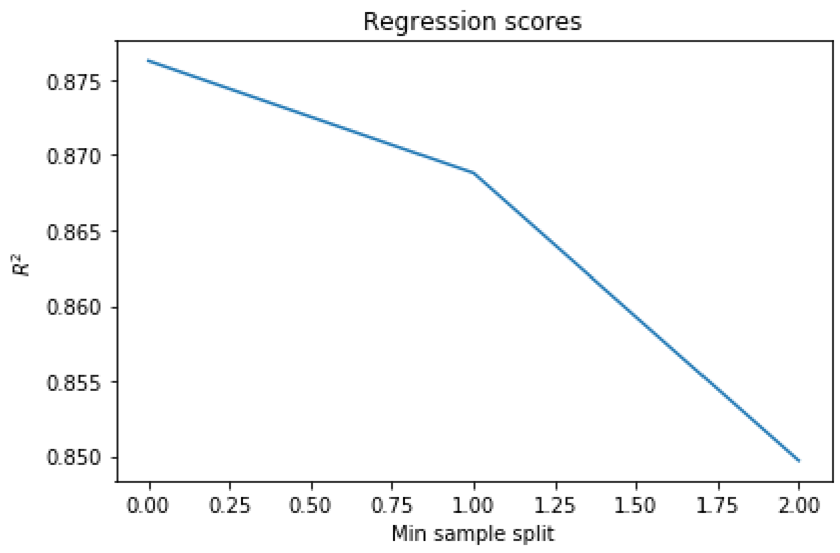


Min sample split = 20
$R^2$ = 0.849709



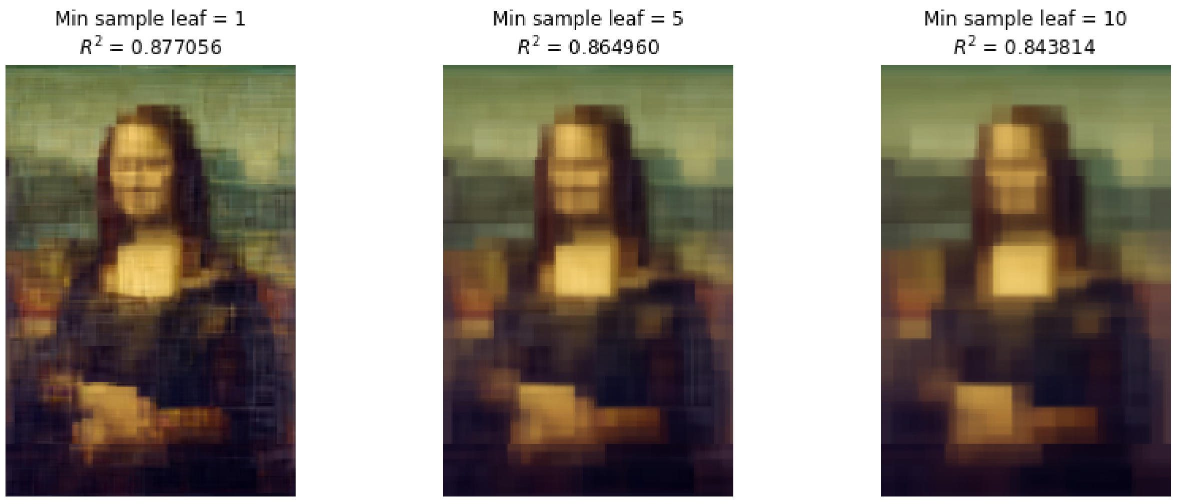
In [12]:

```
plt.plot(regression_scores)
plt.title('Regression scores')
plt.ylabel('$R^2$')
plt.xlabel('Min sample split')
plt.tight_layout()
```



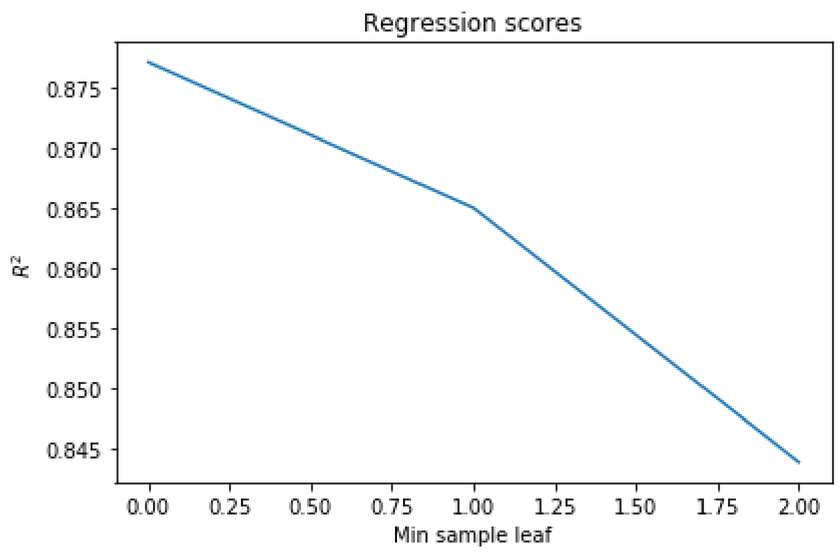
In [13]:

```
# Prunning by min_samples_leaf
leaf_list = [1, 5, 10]
n_trees = 50
max_depth = 20
regression_scores = []
plt.figure(figsize=[14, 5])
for i, leaf in enumerate(leaf_list):
    reg = RandomForestRegressor(n_estimators=n_trees, max_depth=max_depth, min_samples_leaf
    reg.fit(x_tr, y_tr)
    pred = reg.predict(x_te).reshape(pic.shape)
    plt.subplot(1, 3, i+1)
    plt.imshow(pred)
    plt.axis('off')
    sc = reg.score(x_te, y_te)
    regression_scores.append(sc)
    plt.title('Min sample leaf = %d\n$R^2$ = %f' % (leaf, sc))
```



In [14]:

```
plt.plot(regression_scores)
plt.title('Regression scores')
plt.ylabel('$R^2$')
plt.xlabel('Min sample leaf')
plt.tight_layout()
```



For prunning, I set no of trees to be 50 and depth to be 20. I used two prunning methods i.e., min_samples_split and min_samples_leaf. We see for both, prunning is making it worse; that implies our model is robust and good.

f: Analysis

i. The decision rule at each split point could be defined as the following: Next node assignment = left_node if x >= some_threshold = right_node, otherwise Basically, our input is the (x,y) coordinates and we output either a node on the next level of the decision tree, or in the case of the leaf node we output the acutal pixel (R,G,B) value.

ii. The resulting images look like square patches of color. This is due to the fact that random forest approach segments the image into rectangle bins and assigns colors, giving them a patchy look.

iii. For a single decision tree the number of patches of color will be 2^depth. Each leaf should result in a patch of color. This is due to the fact we only have one tree the maximum number of patches will be 2^depth.

iv. For n decision tree, the maximum number of leaves, which is patches of color in this case, should be $n \times 2^d$, where d represents the depth of decision trees.

References:

1. https://github.com/WeimingZhang/ (https://github.com/WeimingZhang/)
2. https://github.com/sidxiong/ (https://github.com/sidxiong/)

In collaboration with rmm223

```
In [ ]:
```