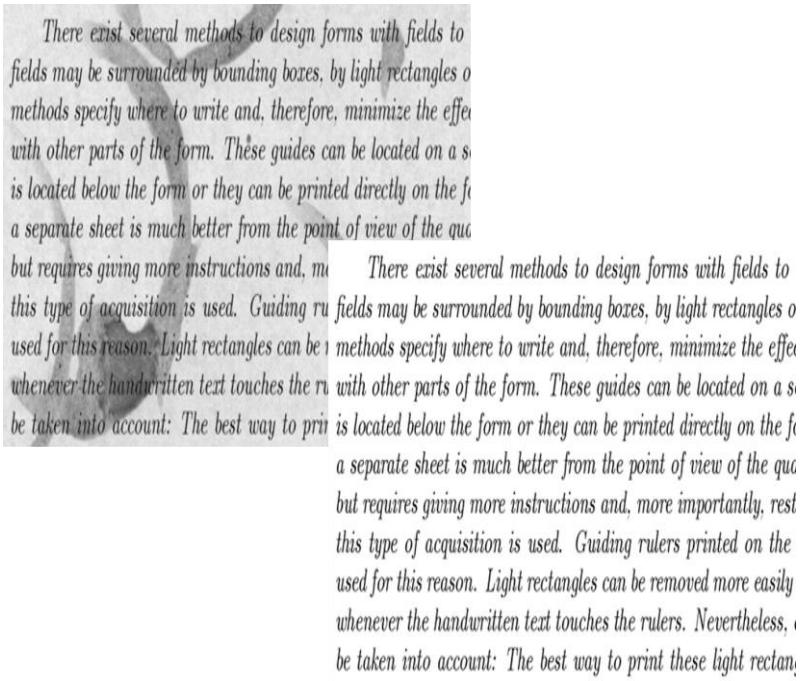


DENOISING DIRTY DOCUMENTS

Team smaRt

Deepak Sisodia
Mir Akram Ali Yadullahi
Harsha Chandar
Pranav Sachdev
Shaikh Shaikh

Table of Contents

EXECUTIVE SUMMARY

The heart and soul of business operations of almost any organization is largely concentrated on electronic, digital and paper documents, which are created and distributed in real-time throughout the organization. Optical Character Recognition (OCR) is a process of getting type or hand written documents into a digitalized format. However, a major hurdle in digitalizing these documents is the presence of noise in one of the following forms – coffee stains, faded sunspots, dog-eared pages, lot of wrinkles etc. Our objective is to develop a machine-learning algorithm that can cleanse such images and remove the noise from the text. Our team implemented 4 independent image processing algorithms – Edge Detection Techniques, Adaptive Thresholding, Median Filtering and Feature Engineering – in order to 'denoise' the dirty documents and extract the cleaned version of the image. Each of these independent algorithms was then combined to develop an ensemble machine-learning algorithm – XGBoost, by training the algorithm on a collection of 'noisy' images. The XGBoost model developed had a 'root mean squared error' value of 3%, showing a marked improvement over the individual image processing algorithms and extracted the text without the noise.

INTRODUCTION

Optical Character Recognition (OCR) is a technique for converting typed, handwritten or printed text into machine-encoded text. OCR makes previously static content editable, searchable, and much easier to share. But, a big constraint in the way of digitizing documents is the presence of noise in the text images. Noise in the form of coffee stains, faded sun spots, dog-eared pages, and wrinkles is frequently present in the text images.

1.1 OBJECTIVE

Given a series of dirty images and a series of clean images, the objective is to create a predictive algorithm that uses image processing and machine learning approaches to clean up the noisy images of text and get us from the dirty images to the clean images.

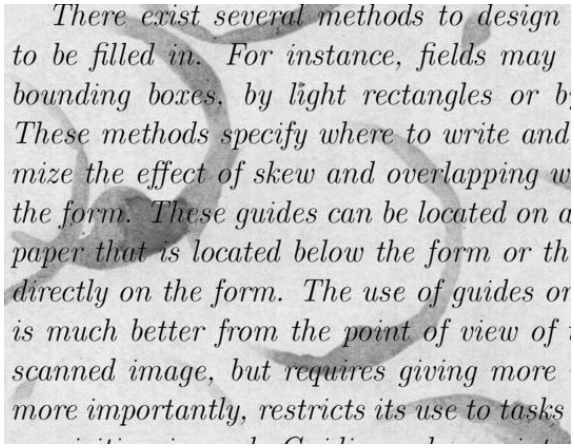
1.2 BUSINESS USE CASE

The enhancement of the text images would help OCR technique to successfully and accurately convert text images to live text. Using this technique, a lot of old and rare documents as well as day-to-day official documents such as receipts, memos, etc.… can be digitized.

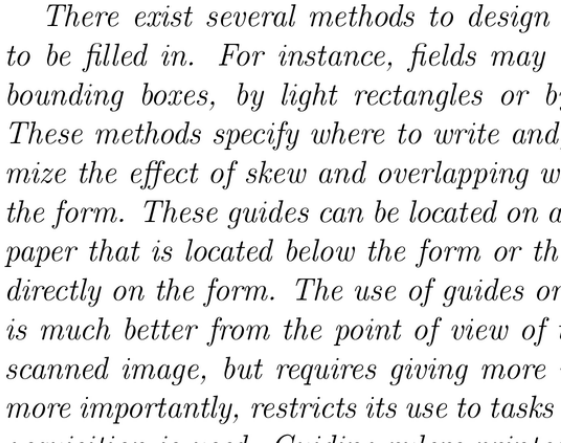
1.3 DATASET

The input dataset has two sets of images, dirty and clean. Dirty images contain various styles of text, to which synthetic noise has been added to simulate real world, messy artifacts. Clean images do not contain any noise and provide us the expected output for each corresponding dirty image.

Dirty images are grayscale images whereas clean images are black and white. This is because any kind of background has been removed from the grayscale dirty images. Shown below is a sample dirty image along with its corresponding clean image.



Sample dirty input text image



Sample clean text image

2 IMAGE DE-NOISING APPROACHES

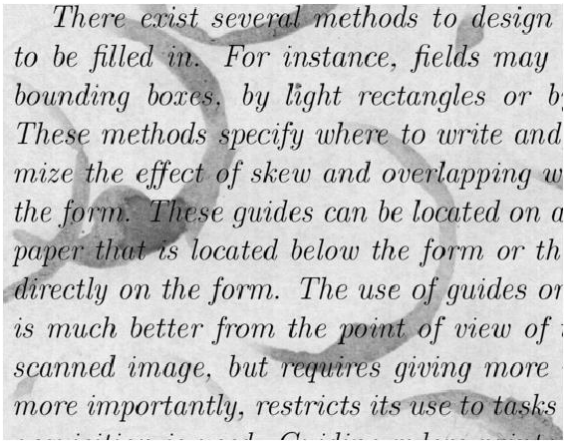
2.1 EDGE DETECTION AND MORPHOLOGICAL TECHNIQUES

2.1.1 Overview

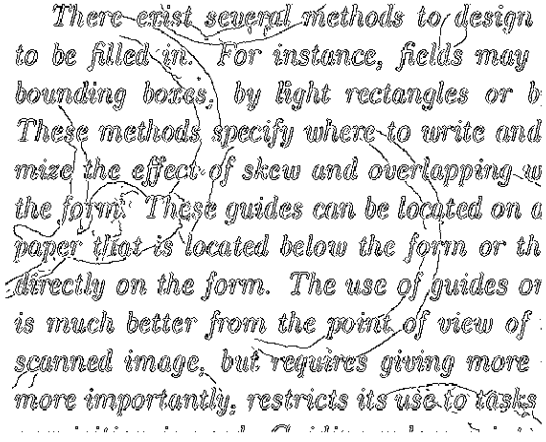
To remove noise (like coffee stains etc.) from the input scanned document image, we exploited the fact that strokes of writing have a pattern where the edges are parallel and space between the two edges is thin and fully filled with black pixels. The noise in the form of coffee stains is random, thick, and lacking any pattern. In addition, the edges of the stains are not parallel and fill inside the stain is non-uniform. The noise in the document has a discontinuous pixel intensity whereas a writing stroke has a uniform pixel intensity. We used this information to distinguish stain from the text.

2.1.2 Edge Detection

Firstly, we performed edge detection on the input image. The edge detection process removed the discontinuous fill inside the stain and only kept edges of the objects. Edge detection on the text will change the black fill text to outlined text without any fill.



Input Grayscale image



B&W output image after Edge Detection

Two things are worth noting in the above images-

1. The input image is grayscale but output after edge detection is black and white.
2. After edge detection, only the edges of the stains remained on the image and fill inside the stain is removed. Also, fill inside text characters is removed resulting in outline appearance of the text.

2.1.2.1 Edge Detection Algorithm

A grayscale image is actually a collection of pixels arranged in rows and columns like a matrix. Each pixel has some pixel intensity. A grayscale image is a 256-bit image which means there are total 256-pixel intensity levels i.e. pixel

intensity values from 0 through 255. 0 corresponds to black color and 255 corresponds to a white color. All values less than 256 and greater than 0 are different shades of gray.

Instead of a 0-255 scale, pixel intensity scale can be changed to 0-1 scale by dividing all pixel intensity values by 255. On this 0-to-1 scale, 0 means black whereas 1 means white. All values between 0 and 1 are different shades of gray.

Mathematically, an image can be represented as a matrix where each cell has some value corresponding to the intensity or brightness of the particular pixel. All cell values lie in range (0,1) and there will be 256 discrete levels between 0 and 1. For Example, a grayscale image can be represented on a 0-to-1 scale in the form of a matrix shown below-

0.9	0.60	0.01	0.44	0.88
0.89	0.90	0.90	0.88	0.86
0.9	0.91	0.89	0.89	0.89
0.12	0.3	0.60	0.88	0.89
0	0.28	0.55	0.1	0.51
0.89	0.89	0.9	0.89	0.80
0.91	0.89	0.88	0.42	0.52

Following are the steps we followed for implementing edge detection-

Step 1 - Edges in an image are the points where intensity change (increase or decrease) is maximum. So in the above matrix, we calculated the change in intensity in x-direction as well as in y-direction.

Intensity change at each pixel in x direction = $\frac{1}{2}*(X_{i,j} - X_{i,j-1}) + \frac{1}{2}*(X_{i,j} - X_{i,j+1})$
Intensity change at each pixel in y direction = $\frac{1}{2}*(Y_{i,j} - Y_{i-1,j}) + \frac{1}{2}*(Y_{i,j} - Y_{i+1,j})$

Pixel intensity change for the above grayscale image is represented in the form of a matrix as shown below-

0.2	0.445	0.51	0.435	0.28
0.06	0.005	0.01	0.02	0.08
0.055	0.015	0.01	0	0.055
0.53	0.24	0.29	0.145	0.06
0.64	0.275	0.36	0.43	0.45
0.055	0.005	0.01	0.05	0.145
0.055	0.015	0.235	0.28	0.29

Matrix of pixel intensity change in x-direction

0.055	0.35	0.94	0.5	0.07
0.01	0.155	0.45	0.225	0.025
0.395	0.31	0.15	0.01	0.015
0.45	0.315	0.17	0.395	0.19
0.505	0.315	0.2	0.785	0.335
0.455	0.305	0.185	0.63	0.285
0.055	0.055	0.07	0.525	0.38

Matrix of pixel intensity change in y-direction

Step 2- Set a threshold level and all points above this threshold are the points lying on edges and hence assigned value 0 and rest others 1.

Threshold level = 0.5
If change in intensity >= 0.5 then set pixel intensity = 0 (i.e. pixel lying on edge)
If change in intensity < 0.5 then set pixel intensity = 1 (i.e. pixel not lying on edge)

Step 3- Perform AND operation on pixel intensity change matrices in x-direction and y-direction.

1	1	0	0	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
0	1	1	0	1
1	1	1	0	1
1	1	1	0	1

Pixel intensity change in x-direction after thresholding

1	1	0	1	1
1	1	1	1	1
1	1	1	1	1
0	1	1	1	1
0	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Pixel intensity change in y-direction after thresholding

The final output matrix after AND operation is shown below. It represents a black and white image (only 0s and 1s) having black color pixels only on the object edges.

1	1	0	0	1
1	1	1	1	1
1	1	1	1	1
0	1	1	1	1
0	1	1	0	1
1	1	1	0	1
1	1	1	0	1

B&W image pixel representation after edge detection

2.1.2.2 Edge detection implementation

The above algorithm for edge detection is a very crude one. Its output was not perfect. Hence we used an image processing package in R 'biOps' to use an advanced algorithm for edge detection named 'Canny Edge Detection'.

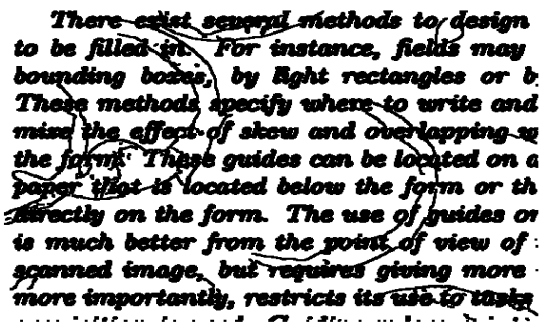
'biOps' package has been removed from CRAN and moved to google archive. To install 'biOps' package follow the instructions given here. One thing to note is that biOps images have pixel intensities from 0 to 255 rather than from 0 to 1. So before using any biOps package function we have to change the pixel intensities by multiplying the intensities by 255.

Below R code snippet will run edge detection on the input image (image.matrix)

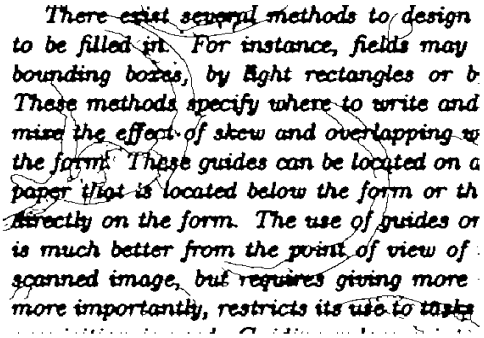
```
imagedata.inputimage=imagedata(image.matrix*255)
imagedata.edgedetected = imgCanny(imagedata.inputimage, 0.7)
```

2.1.3 Morphological Techniques

In a morphological operation, each pixel in the input image is compared with its neighbors and a value of each pixel in the output image is assigned. The most basic morphological operations are dilation and erosion. Dilation adds pixels to the boundaries of objects in an image, while erosion removes pixels on object boundaries. The number of pixels added or removed from the objects in an image depends on the size and shape of the mask or structuring element used to process the image. In our case, we have chosen a 3x3 square matrix as our structuring element or mask.



Edge-detected image after pixel dilation



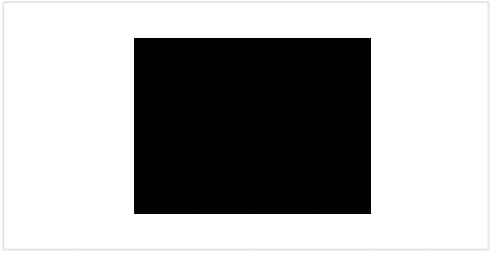
Edge-detected + dilated image after pixel erosion

Dilation and Erosion algorithms

The input to dilation and erosion algorithm in our case is a black and white image (containing only 0s and 1s) obtained after edge detection stage. For the sake of convenience let's assume that the image after edge detection is in the below matrix format. This matrix is actually representing a black fill rectangle surrounded by white pixels.

1	1	1	1	1	1	1	1
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	1	1	1	1	1	1

Matrix representation of an image



Pixel representation of an image

Pixel Dilation Algorithm

For each white pixel in the input image, all 8 surrounding pixels are checked. If at least one pixel has value 0 then input white (1) pixel is changed to black (0).

1	1	1	1	1	1	1	1
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	1	1	1	1	1	1

1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	1	1	1	1	1	1

This process is done simultaneously for all white pixels in the input image. The final output of the dilation process for the above image matrix is shown below.

1	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1

Matrix representation of dilated image

Pixel representation of dilated image

Pixel Erosion Algorithm

For each black pixel in the input image, all 7 surrounding pixels are checked. If at least one pixel has value 1 then input black (0) pixel is changed to white (1).

1	1	1	1	1	1	1	1
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	1	1	1	1	1	1

1	1	1	1	1	1	1	1
1	1	1	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	1	1	1	1	1	1

This process would be done simultaneously for all black pixels in the input image. The final output of erosion process for the above image matrix is as shown below-

1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	0	0	1	1	1
1	1	1	0	0	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

Matrix representation of eroded image

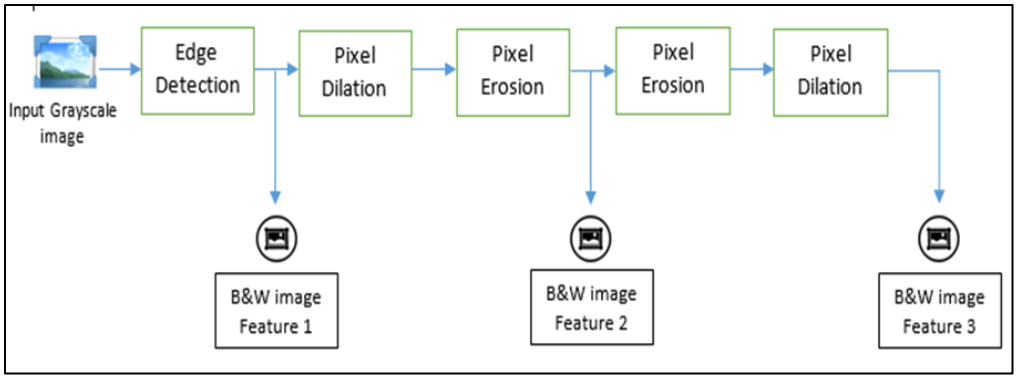
Pixel representation of eroded image

2.1.4 Pixel Dilation and Erosion Implementation

We created two functions in our R script namely pixelErosion() and pixelDilation() to erode and dilate image pixels respectively. Refer appendix to view the code for these functions.

2.1.5 End to End Block Diagram

Using the above-described three techniques we created 3 additional features which would act as an independent variable to our prediction model.



2.2 MEDIAN FILTERING

2.2.1 Overview

The **median filter** is a nonlinear digital filtering technique, often used to remove noise. Median filtering is very widely used in digital image processing because, under certain conditions, it preserves edges while removing noise. A median filter is an image filter that replaces a pixel with the median value of the pixels surrounding it. In doing this, it smoothes the image, and the result is often thought of as the "background" of the image, since it tends to wipe away small features, but maintains broad features.

2.2.2 Algorithm

The central theme of implementing a median filter is to run through the image, pixel by pixel, replacing each pixel value with the median of the neighboring entries. The pattern of the neighbors is called as the 'window', which slides pixel by pixel over the entire image. There are two types of median filter implementations:

Demonstration

To demonstrate, using a window size of 3, with one entry immediately preceding and following each entry, a median filter will be applied to the following simple 1D matrix/signal.

a. Consider a 1-dimensional signal as below

$$\text{Signal\_1d} = [6 \quad 2 \quad 14 \quad 25]$$

b. The median output filtered output signal y will be:

$$Y[1] = \text{Median}[6 \quad 6 \quad 2] = 6$$

$$Y[2] = \text{Median}[6 \quad 2 \quad 14] = 6$$

$$Y[3] = \text{Median}[2 \quad 14 \quad 25] = 14$$

$$Y[4] = \text{Median}[14 \quad 25 \quad 25] = 25$$

$$\text{Hence, } Y = [6 \quad 6 \quad 14 \quad 25]$$

Note that, because there is no entry preceding the first value, the first value is repeated, as with the last value to obtain enough entries to fill the window.

Median Filter Algorithm

1. Consider the following matrix $\begin{bmatrix} 5 & 4 & 8 \\ 2 & 1 & 9 \\ 13 & 3 & 11 \end{bmatrix}$

```
img = matrix(c(5,2,13,4,1,3,8,9,11),nrow=3,ncol = 3)
img
```

```
> img
     [,1] [,2] [,3]
[1,]    5    4    8
[2,]    2    1    9
[3,]   13    3   11
```

2. Create an empty output matrix of the same size as that of the input matrix (in this case, a 3*3 matrix) as follows

```
#Create an empty output matrix of the same size as the input image
Y = matrix(0,nrow(img),ncol(img))
Y
```

```
> Y
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0
>
```

3. Let us consider a window width of size 3. Pad the input matrix with zeros on all the sides (for a window of size k, pad rounded down value of k/2 rows and columns with 0). This can be done in two steps as follows

   2.a: For a window size of 3, we will first define a new matrix with all 0 values, whose dimensions are as follows

```
k = 3
n = floor(k/2)
#Modify the input image matrix by padding 0s outside the input image matrix
#to make it size having an additional n rows and n columns
img_modify = matrix(0,nrow(img)+2*n,ncol(img)+2*n)
img_modify
```

```
> img_modify
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0
[4,]    0    0    0    0    0
[5,]    0    0    0    0    0
```

2.b: Place the input matrix (img), in the center of this newly created padded matrix as follows

```
#Copy the Original matrix to the zero/padded matrix
row_seq = seq(nrow(img))
col_seq = seq(ncol(img))

for (x in row_seq)
{
  for (y in col_seq)
  {
    img_modify[x+n,y+n] = img[x,y]
  }
}
img_modify
```

```
> img_modify
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    5    4    8    0
[3,]    0    2    1    9    0
[4,]    0   13    3   11    0
[5,]    0    0    0    0    0
>
```

4. Consider a window of size 3 by 3. The window can be of any size. Starting from the 3*3 matrix, from the first row and the first column – "img_modify [1,1]". This matrix, called as the 'window' is as below

$$\text{window} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 5 & 4 \\ 0 & 2 & 1 \end{bmatrix}$$

5. Calculate the median value of the matrix – window. This is the middle most value after sorting the window matrix

$$\text{Sort of the window} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \mathbf{0} & 1 \\ 2 & 4 & 5 \end{bmatrix}$$

The median value is the middle most value – which is '0' as highlighted above. This value of the median goes in as the first member of the output matrix Y

6. This procedure is repeated for all the values in the input matrix by sliding the window to the next partition i.e. starting from the first row and second column – img_modify [1,2]. This process is followed iteratively. The following code achieves this.

```
for (i in seq(nrow(img_modify)-k))
{
  for (j in seq(ncol(img_modify)-k))
  {
    window = matrix(0,k*k,1)
    c = 1
    for (x in seq(k))
    {
      for (y in seq(k))
      {
        window[c] = img_modify[i+x-1,j+y-1]
        c = c + 1
      }
    }

    med = sort(window)

    Y[i,j] = med[((k*k)+1)/2]
  }
}
```

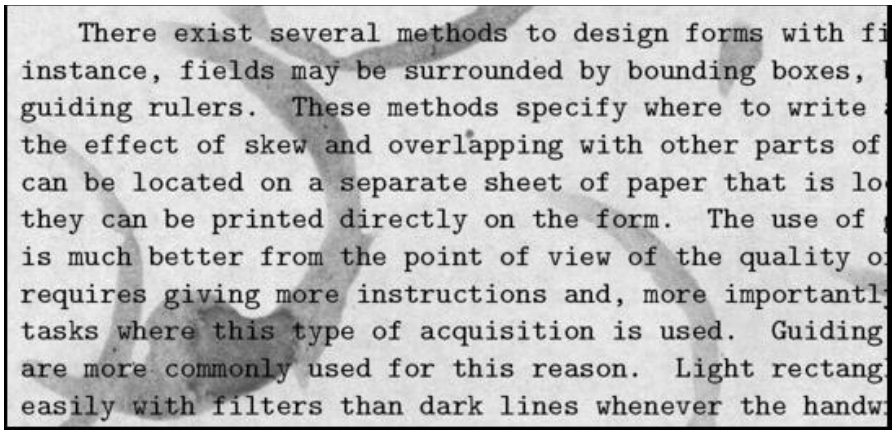
The final median filter output = Y, is shown below

```
> Y
     [,1] [,2] [,3]
[1,]    0    2    0
[2,]    2    5    0
[3,]    0    0    0
>
```

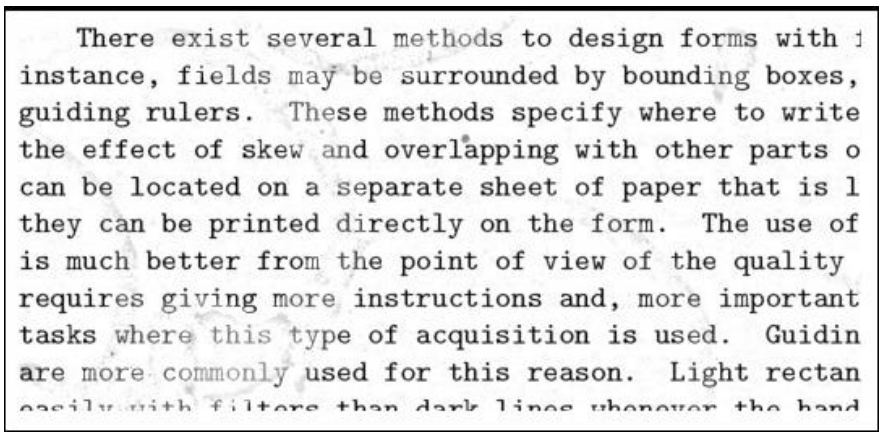
2.2.3 Implementing 2D Median Filtering to Image Processing and Noise Reduction*:*

The 2D median filter algorithm that we developed above, was applied to image processing in order to remove noise from the images. The implementation smoothes the image, and the result can be considered as the "background" of the image, since it tends to wipe away small features, but maintains broad features. A sample of the input image and the cleansed output image (after processing the median filter algorithm) can be seen below

Raw Image (With Noise)



Cleansed Image (Implementing the Median Filter Algorithm) – with a window size of 9


There exist several methods to design forms with i
instance, fields may be surrounded by bounding boxes,
guiding rulers. These methods specify where to write
the effect of skew and overlapping with other parts o
can be located on a separate sheet of paper that is l
they can be printed directly on the form. The use of
is much better from the point of view of the quality
requires giving more instructions and, more important
tasks where this type of acquisition is used. Guidin
are more commonly used for this reason. Light rectan
easily with filters than dark lines whenever the hand

Implementation Steps in R

1. The input to our algorithm is a text image file, to which some noise has been added. In the case of the sample image we processed, this noise was in the form of a coffee stain. We also have a corresponding clean image, which we will use in order to determine the strength of our image processing model/algorithm.

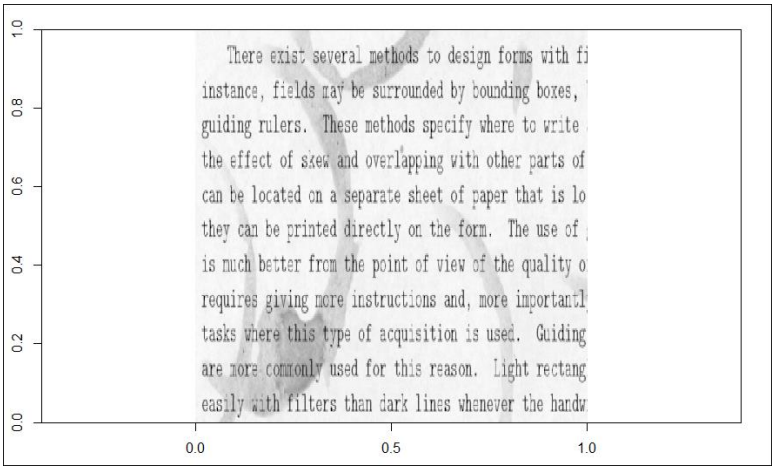
2. When an image file is read in R, it is stored in the form of a 2-dimensional array, with the rows and columns showing the image positions and the members of the array showing the pixel values. We can use the 'png' package in R in order to read the input image as follows

```
img = readPNG("C:/Users/mirak/Documents/Classes_Fall_16/R/Project 1/train/62.png")
CleanImg = readPNG("C:/Users/mirak/Documents/Classes_Fall_16/R/Project 1/train_cleaned/62.png")
```

   'img' corresponds to the input image with noise (which has to be processed). The 'CleanImg' is the completely cleaned version of the image, which is used to assess the accuracy of our algorithm

3. We can plot the image in the form of a heat-map, showing the variations in the brightness/image pixels using the function – 'raster'

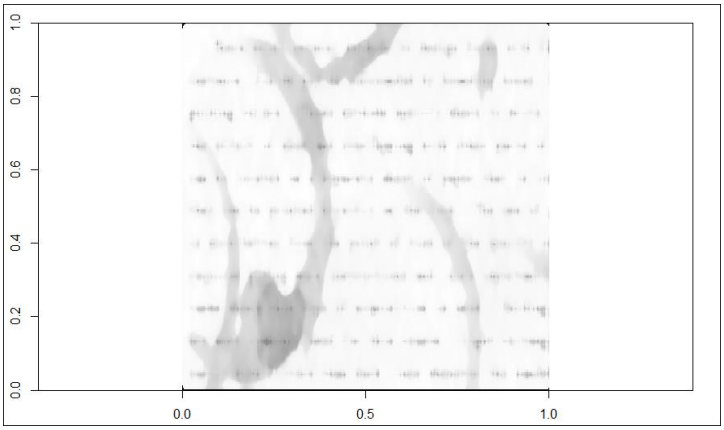
```
plot(raster(img),col = gray.colors(256,start=0,end=1,gamma=2.2,alpha=NULL))
plot(raster(CleanImg),col = gray.colors(256,start=0,end=1,gamma=2.2,alpha=NULL))
```



This plot shows the input image, with the brightness in the range of [0, 1], with 0 being black and 1 being white.

4. The above image is passed as an input matrix to the 2D median algorithm described in the above section (2.2 2D Median Filter Algorithm). We now used a window size of k=9.

5. As we pass the input image matrix to the 2D median filter, the output matrix is the image background (white background along with the noise).



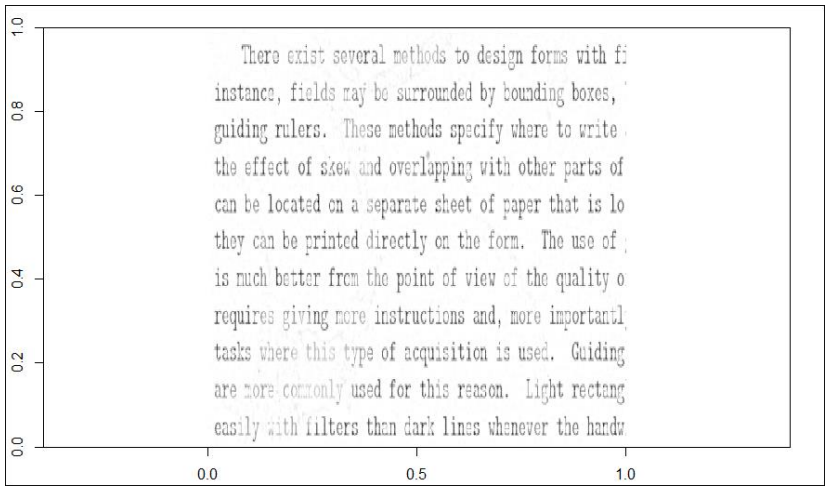
6. The final step is to extract the foreground (actual text) of the image. This can easily be achieved by subtracting the background image from the original input image. This is achieved in two steps as below:
   a. Take the difference between the background image and the original input image. Also, since the writing is always darker than the background, the foreground should only show pixels that are darker than the background.

```
foreground = img - background
#In this case, we know that the writing is always darker than the background,
#so, our foreground should only show pixels that are darker than background
foreground[foreground > 0] = 0
```

b. The foreground should also be rescaled to contain values between [0, 1]. This can be achieved through a simple normalization as below

```
#Normalizing the final results (pixels) to lie between 0-1
m1 = min(foreground)
m2 = max(foreground)

foreground = (foreground - m1) / (m2 - m1)
```

7. We get our final cleansed image as follows



8. We also calculate the 'Root Mean Squared Error' value in order to determine the degree of accuracy of conversion, as follows

```
#Calculate the RMSE

rmse = sqrt(mean((foreground - cleanImg)^2))
print(c(rmse,k))
```

```
>   print(c(rmse,k))
[1] 0.08788531 9.00000000
>
```

We note that the RMSE is 8.7%.

The algorithm developed was converted to a function, which takes the input image and the window size as the inputs and gives the cleansed image as an output. The output is used as one of the regressors in developing the final machine learning algorithm.

2.3 ADAPTIVE THRESHOLDING

2.3.1 Overview

Thresholding is a simple way to segregate the objects from the background. Global thresholding can be used to binarize the pixel intensity if we have a relatively uniform background. Adaptive thresholding can be used when there is a large variation in the background intensity. There are different types of adaptive thresholding

Adaptive Thresholding technique	R package (function)
Empirical Bayes thresholding	EbayesThresh (ebayesthresh)
Tree based thresholding	treethresh (treethresh)
wavelet thresholding	treethresh (threshold)
Local adaptive	EBImage (thresh)

2.3.2 Algorithm

Adaptive thresholding works by comparing each pixel's intensity to the background determined by local neighborhood. "thresh" function in package EBImage provide us with an adaptive thresholding using a linear filter with a rectangular box. Syntax for adaptive thresholding is

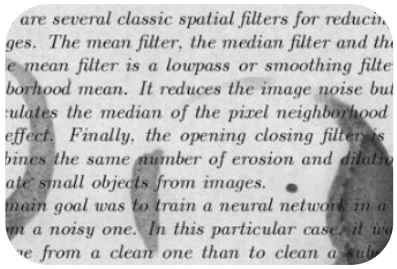
thresh(x, w=5, h=5, offset=0.01)

x: Image object,

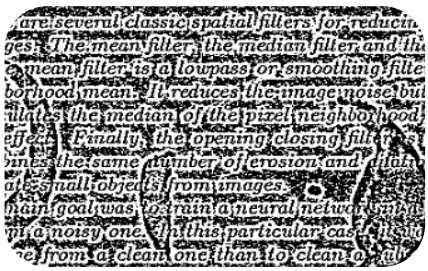
w, h: width and height of the moving rectangular window.

Offset: Thresholding object from the average value

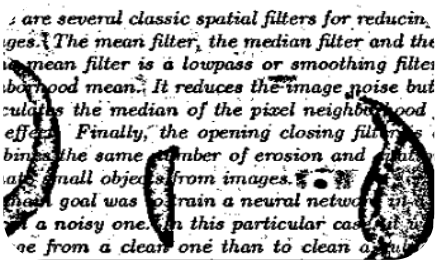
Output images with different width and height



Original



w=2 , h=2



w = 10, h =10

We can observe that function is keeping the image but adding some unwanted noise to the background. We can combine different thresholded images along with original image and use maximum pixel brightness.

The following function can be used to choose the pixel among different images

Img: image that needs to be processed
img2Mat: convert image to matrix
img2vec: convert image to vector

```
Adaptive_thresholding <- function () {
Threshold_image.2 <- thresh (img, 2, 2) // Adaptive thresholding with different width and height
Threshold_image.5 <- thresh (img, 5, 5)
Threshold_image.7 <- thresh (img, 7, 7)
Threshold_image.10 <- thresh (img, 10, 10)

// matrix with all thresholded images
threshold_matrix = cbind (img2vec(Image2Mat(Threshold_image.2)), img2vec(Image2Mat(Threshold_image.5)),
img2vec(Image2Mat(Threshold_image.7)),
img2vec (Image2Mat (Threshold_image.10)))
// get the maximum pixel
max_matrix = apply (threshold_matrix, 1, max)
// form a final matrix
final_matrix = matrix (max_matrix, nrow (img), ncol (img))
}
```

2.4 FEATURE ENGINEERING - IDENTIFYING THE GAPS BETWEEN LINES OF TEXT:

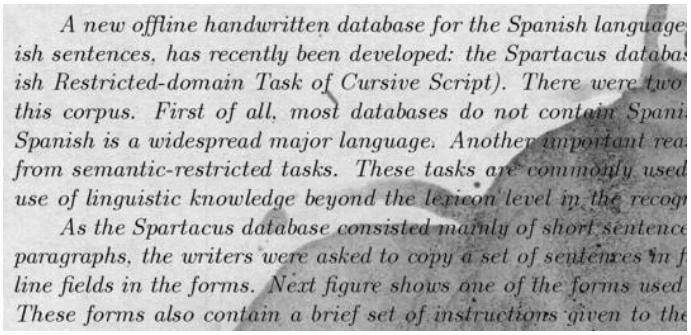
2.4.1 Overview:

There's a very peculiar property of documents which separates them from other images (such as pictures) is the presence of text in the form of sentences. The sentences are linear in nature and form the different rows of the document. The text is arranged into lines, and that there are gaps between those lines of text. If we can find the gaps between the lines, we can ensure that the predicted value within those gaps is always the background color (white).

2.4.2 Assumption:

i. The image is grayscale.
ii. The image is having only two contrast values, the text is black (pixel value of 0) and the background is white (pixel value of 1).

Input:



A new offline handwritten database for the Spanish language
ish sentences, has recently been developed: the Spartacus databa
ish Restricted-domain Task of Cursive Script). There were two
this corpus. First of all, most databases do not contain Spani
Spanish is a widespread major language. Another important rea
from semantic-restricted tasks. These tasks are commonly used
use of linguistic knowledge beyond the lexicon level in the recog
As the Spartacus database consisted mainly of short sentence
paragraphs, the writers were asked to copy a set of sentences in f
line fields in the forms. Next figure shows one of the forms used
These forms also contain a brief set of instructions given to the

2.4.3 Approach:

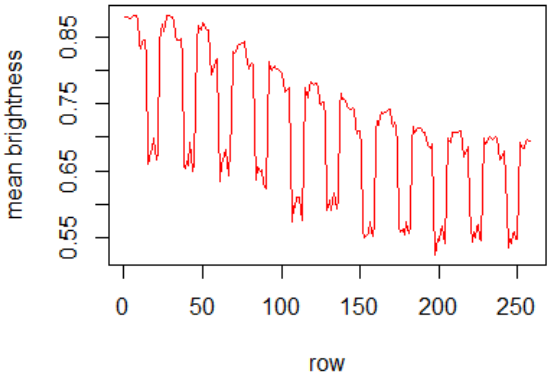
i. Load the dirty image (with noise) in R by using PNG package. The image is loaded as a matrix with pixel values stored for each value of row and column.

```
> dim(imgX)
[1] 258 540
> nrow(imgX)
[1] 258
> ncol(imgX)
[1] 540
```

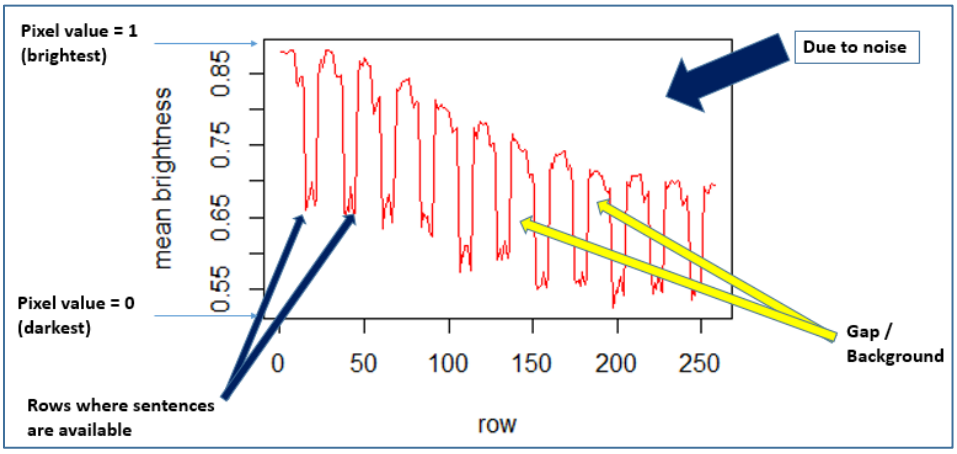
The generated matrix of the sample image has 258 rows and 540 columns, and the matrix containing the pixel values looks like:

```
        [,533]    [,534]    [,535]    [,536]    [,537]    [,538]    [,539]    [,540]
 [1,] 0.87450980 0.8823529 0.85490196 0.86666667 0.86666667 0.8823529 0.8862745 0.88627451
 [2,] 0.90588235 0.9098039 0.89019608 0.85098039 0.89803922 0.8901961 0.8941176 0.90588235
 [3,] 0.87843137 0.8784314 0.88235294 0.89411765 0.89803922 0.8941176 0.8980392 0.87058824
 [4,] 0.86666667 0.8784314 0.88627451 0.87450980 0.88627451 0.8784314 0.8823529 0.87450980
 [5,] 0.86666667 0.8745098 0.87843137 0.87058824 0.88627451 0.8784314 0.8745098 0.86274510
 [6,] 0.87450980 0.8745098 0.88235294 0.90196078 0.89411765 0.8823529 0.8784314 0.85490196
 [7,] 0.85098039 0.8549020 0.86274510 0.88627451 0.87843137 0.8745098 0.8705882 0.84705882
 [8,] 0.85490196 0.8784314 0.89411765 0.84313725 0.84705882 0.8705882 0.8666667 0.85098039
 [9,] 0.87450980 0.8941176 0.90196078 0.86666667 0.87058824 0.8941176 0.8980392 0.85882353
[10,] 0.86666667 0.8588235 0.85882353 0.87058824 0.87450980 0.8705882 0.8588235 0.83921569
[11,] 0.86274510 0.8666667 0.86666667 0.87843137 0.88235294 0.8862745 0.8509804 0.85490196
[12,] 0.84705882 0.8431373 0.85490196 0.86274510 0.86274510 0.8627451 0.8470588 0.84313725
[13,] 0.83529412 0.8313725 0.85490196 0.87450980 0.88627451 0.8666667 0.8549020 0.82352941
[14,] 0.84313725 0.8588235 0.80000000 0.75686275 0.79215686 0.8549020 0.8627451 0.85098039
[15,] 0.79607843 0.5882353 0.50588235 0.54509804 0.45098039 0.4705882 0.7921569 0.72549020
[16,] 0.42352941 0.3725490 0.79215686 0.87843137 0.72549020 0.2274510 0.6588235 0.57647059
[17,] 0.09411765 0.5843137 0.75294118 0.68627451 0.41568627 0.4745098 0.6666667 0.56470588
[18,] 0.14901961 0.4980392 0.51764706 0.55294118 0.63921569 0.8078431 0.6901961 0.54901961
```

ii. In order to compute the location of a row where the sentence is available, we compute the row-wise mean pixel values for the above matrix.



The above plot of mean pixel values for each row is explained below:



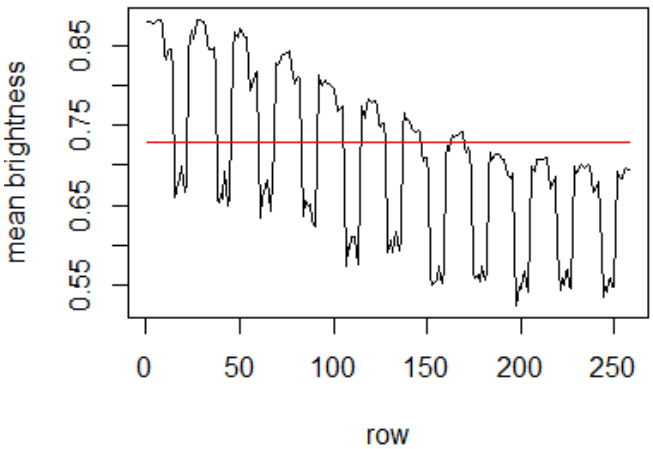
a. The downward trend is due to the presence of noise in the image.
b. The lowest spikes indicate the mean pixel value for the particular row as dark, indicating the presence of a sentence.
c. The locations of higher peaks indicate the rows where the mean pixel value for the row is brightest, indicating the presence of gaps or spaces between the lines; where we assume imputation with the background pixel value of '1'.

iii. Clustering:

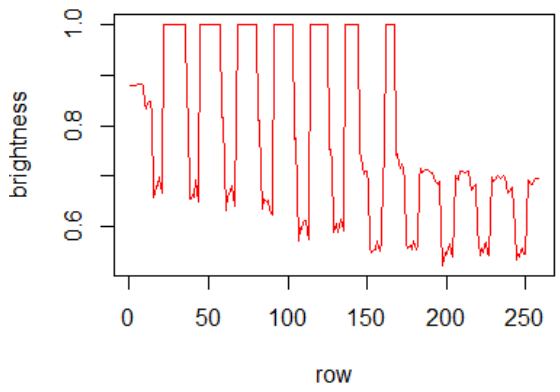
a. Perform K-means clustering with a cluster size of 2 (one for lower mean pixel values, other for relatively higher mean pixel values).
b. Decide the cutoff value.

$$\text{cutoff} = \frac{\text{Highest mean pixel value of lowest cluster} + \text{Lowest mean pixel value of upper cluster}}{2}$$

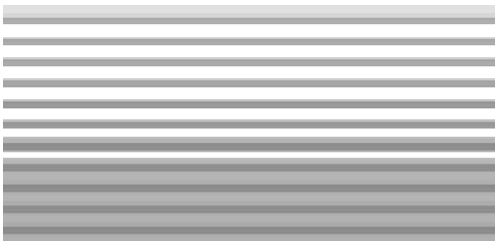
c. The cutoff value for the sample image could be visualized as:



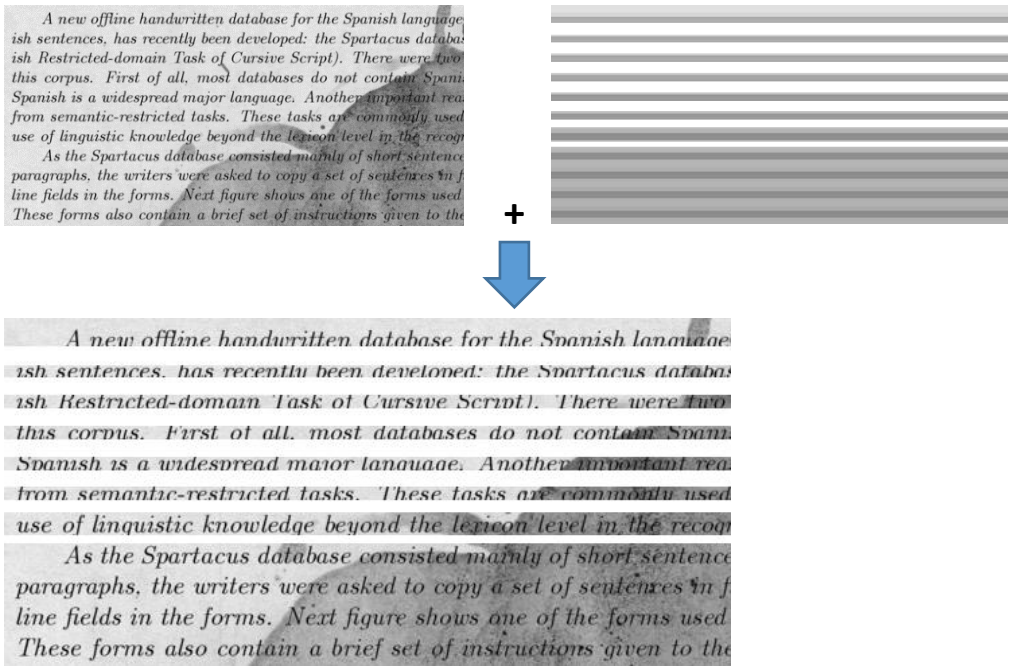
iv. Imputing the gaps which are above the cutoff value with a pixel value of 1 (background pixel value):



v. Looking at the skeleton of the dummy image:



vi. Comparing the skeleton image with the input sample image (having noise), we get the output image:



vii. A major chunk of noise can be removed from the image (subject to the image and the intensity of noise in the image).

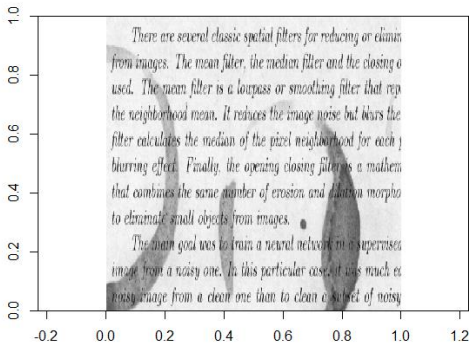
viii. The output vector of this cleaned image is made an input to the XGBoost machine learning algorithm, along with other approaches to predicting the correct pixel values of the sample image with noise.

3 ENSEMBLE OF THE DIFFERENT ALGORITHMS

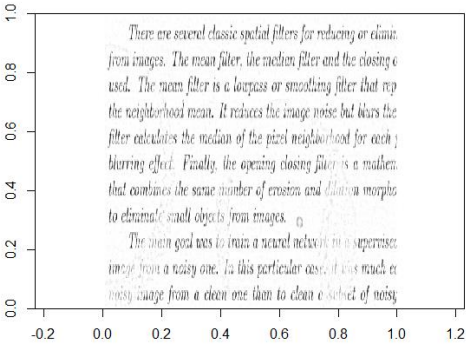
3.1 NEED FOR COMPLETE MODEL

The techniques previously mentioned work with varying efficiency on different types of noise. For example, consider the following raw image.

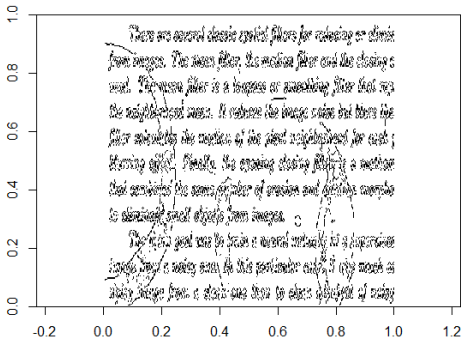
Raw Image:



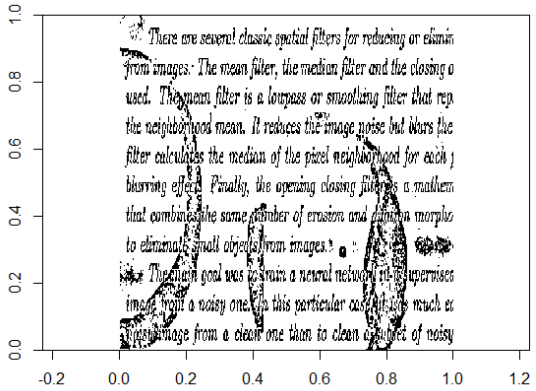
Median Filtering: Almost complete removal of noise but faded line values (Loss of information)



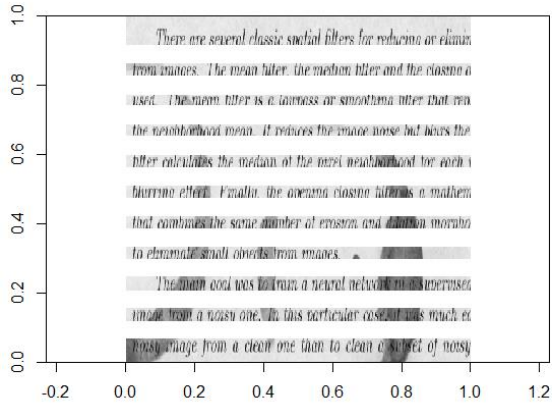
Edge Detection: Distortion of lines



Adaptive Thresholding: Clear lines but incomplete removal of noise



Feature Engineering: Clear lines but incomplete removal of noise



All these results can be combined to give an adequate result, that can be generalized for all types of models.

Therefore, we built an XGBoost model that takes as input, the cleaned images from all these different types of techniques. This supervised learning model is then used to predict the cleaned image.

3.2 GRADIENT BOOSTING:

Gradient Boosting is an ensemble model that tries to improve the prediction power in sequential steps. A simple model is fit on the training data. This model produces some error:

Y = f(x) + e where y is the target variable, f(x) is the model output and e is the error

The next tree is fit on this error:

e = h(x) + e2 where h(x) is the new model, e is the residual from the previous step and e2 is the new error. Another model is fit on this error and the process is repeated till an optimum fit is achieved.

3.3 CROSS VALIDATION:

Cross-validation is a technique that is used to estimate how well the model can be used to predict an unknown dataset. It ensures that problems due to overfitting are reduced.
We used a five-fold cross-validation to determine what was the least number of rounds were necessary to give us the lowest RMSE.
The data is divided randomly into 5 equal sample sets. 4 of the partitioned sets are used for training the model and one partition is used for testing. Five such rounds are repeated and the RMSE is calculated for each round.

3.4 WHY XGBOOST?

XGboost is one of the most efficient implementations of the gradient boosting techniques. Though gradient boosting is a technique that requires sequential processing, XGBoost implements parallelization within a single tree, making the algorithm incredibly fast and suitable for large datasets. It is also widely used, owing to the accuracy of predictions it produces as compared to other algorithms.

3.5 ALGORITHM:

For all images in the data, do the following:
Step 1: Read dirty image & and convert it into a one column matrix

Step 2: Read clean image & convert it into a one column matrix

Step 3: Apply median filtering & convert the cleaned image into a one column matrix

Step 4: Apply feature detection filter & convert the cleaned image into a one column matrix

Step 5: Column bind all these features

Step 6: Append it to the dataframe containing these features for the previous images

Stop loop

Step 7: Take a randomly selected sample of the data (250000 rows were selected)

Step 8: Convert the data into a dense matrix with label being the cleaned image

Step 9: Using cross-validation, determine the optimum number of rounds for the XGBoost model

Step 10: Create the model with RMSE as the evaluation parameter

Step 11: Predict for test images using this model

Step 12: Convert the result into a matrix of the same dimensions as the original image

R implementation:

Load functions from the previous image cleaning steps:
source(file = "ImageCleaning.R")

Though the “apply function” would normally be a faster approach than for loop, for this particular case, for loop had a faster performance (by about 4 minutes). Hence, we are not vectorizing the code.

```
for(file in filenames){
  img = readPNG(file.path(train_folder,file)) #Load image
  clean_img<-readPNG(file.path(target_folder,file)) #Load cleaned image
  x1<-img2vec(img) #Convert image into a single column matrix. This would be the first predictor
  x2<-getLineWidth(img) #Obtain the image after performing featurization to extract line width.
                  #The output of the function is converted into a vector. This would be the second predictor
  x3<-img2vec(median_filter(img,9)) #Obtain the image after performing median filtering.
                        #The result is converted into a single column matrix. This would be the third predictor
  y<-img2vec(clean_img)  #Convert the cleaned image into a single column matrix

imgPred<-rbind(imgPred,cbind(x1,x2,x3,y)) #Combine all the predictors into a dataframe

}
```

The following image processing techniques were deteriorating the quality of prediction and hence removed from the model

```
  #x5<-EdgeDetectionandImageMorphology(img)
  #x6<-img2vec(Adaptive_thresholding(file.path(train_folder,filenames)))
```

To train the model, 250,000 randomly selected values from the above data set were used:

```
 randSample<-sample(nrow(imgPred),250000)
```

The recommended input for an XGBoost model is a DMatrix (Dense Matrix). Hence the data is converted into a DMatrix with the clean image as the label

```
train<-xgb.DMatrix(as.matrix(imgPred[randSample,-5]),label=imgPred[randSample,5])
```

Five fold cross validation is performed to figure out the number of runs required and the XGBoost model is created

```
cv_results = xgb.cv(data = train, nthread = 8, eval_metric = "rmse", nrounds = 500,showsd = T, early.stop.round = 50, nfold = 5,  print.every.n = 10)

#Getting the minimum number of iterations required to get the lowest testing data rmse

min_error_run = which.min(cv_results[, test.rmse.mean])

#Creating the xgboost model

xgb<-xgboost(data=train,nthread=2,nrounds = min_error_run)
```

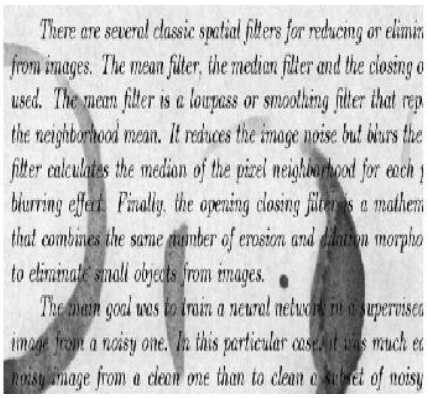
This model is tested on a sample image:

```
#Testing the model with a sample image

#Reading image

pred_Img<-readPNG(file.path(train_folder,filenames[2]))

#Getting all the predictor values for the image using the image cleaning functions

x1<-img2vec(pred_Img)

x2<-getLineWidth(pred_Img)

x3<-img2vec(median_filter(pred_Img,9))

x4<-mean_row_value(pred_Img)

x4<-EdgeDetectionandImageMorphology(pred_Img)

x5<-Adaptive_thresholding(file.path(train_folder,filenames[2]))

#Combining all the predictor values

pred_data<-cbind(x1,x2,x3,x4)

#Naming the columns of the test data

colnames(pred_data)<-c("Original","Featurization","Median Filtering","Mean Row Value")

#Predicting the target values for the test data using the xgb model

pred<-predict(xgb,pred_data)

#Taking care of values outside [0,1]

pred[pred<0]<-0

pred[pred>1]<-1
```

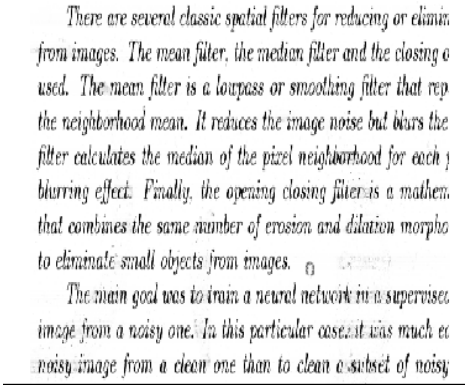
4 CONCLUSION:

4.1 RESULTS

We were able to predict the model with an RMSE of 0.028765 on the training data and 0.031843 on the test data.



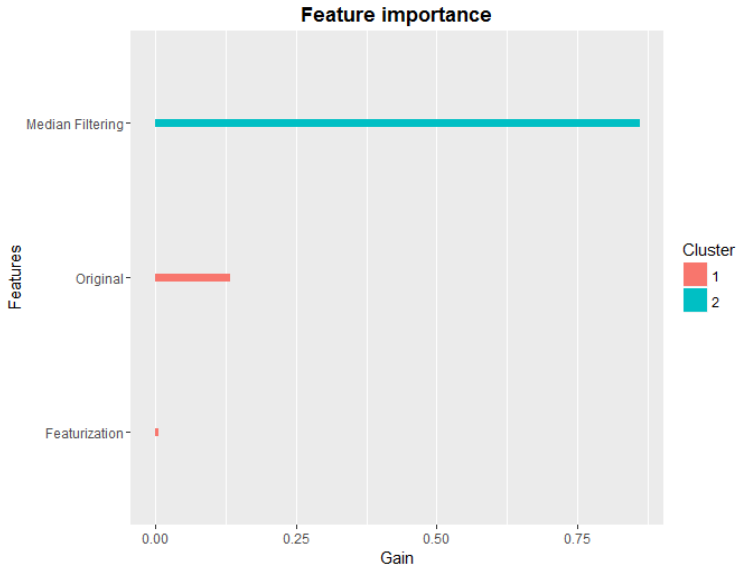
Input Image



Output

The total execution time for training the model using five images is 3.64 minutes.

4.2 FEATURE IMPORTANCE:



Median filtering contributed the most towards getting a clean image, followed by the original image and finally the feature engineering results. The edge detection and thresholding performed poorly and were removed from the model.

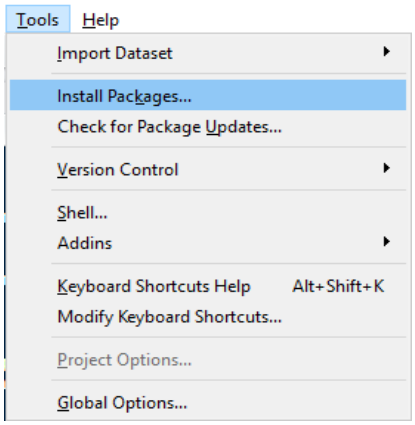
This project was done as a preamble to the optical character recognition process to increase the accuracy of documents that undergo the OCR process. As the next step, we can compare the accuracy of the OCR before and after denoising the image using the model presented in this paper.

5 APPENDIX

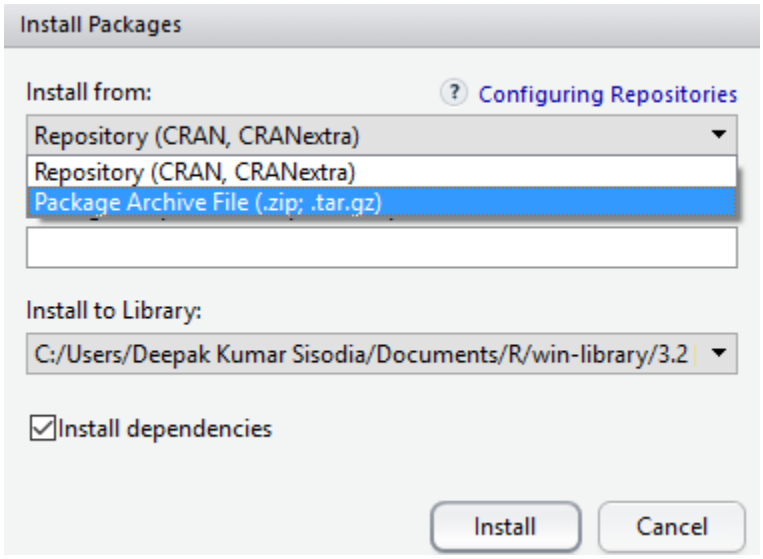
5.1 INSTALLATION INSTRUCTIONS FOR BIOPS PACKAGE ON 64-BIT WINDOWS

The biOps package, which has an implementation of the canny edge detector, has been removed from CRAN. It has been migrated to Google Code. To install biOps package on a 64-bit windows operating system, please follow the below instructions –

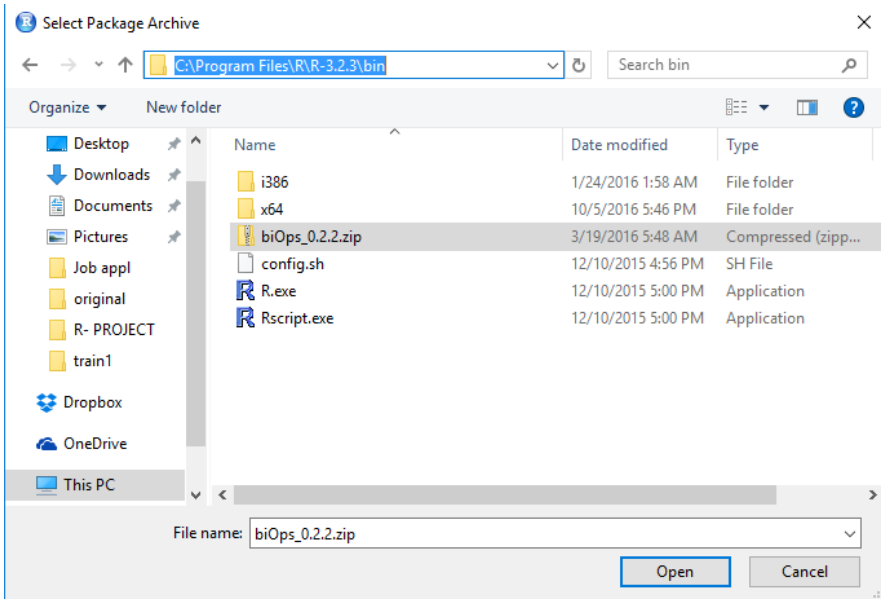
1. Unzip the attached rar file 'biOps.rar'
2. Copy 4 dlls (libfftw3-3.dll, libfftw3f-3.dll, libfftw3l-3.dll, and zlib1.dll) to "C:\Program Files\R\R-3.2.3\bin\x64" folder.
3. Copy 'biOps_0.2.2.zip' file to "C:\Program Files\R\R-3.2.3\bin" folder.
4. Open R Studio running 64-bit R
5. Go to Tools menu item and Select 'Install Packages'



6. Select 'Package Archive File' option from 'Install From:' dropdown.



7. In the browse window, select the 'biOps_0.2.2.zip' file copied in step 3 above.



8. Press Install button
9. After successful installation of biOps package, load the package by running the following command in R Studio -
   library(biOps)

If you are using any other operating system then follow the instructions mentioned here.

Binaries for 64-bit windows operating system are present in the project folder.

5.2 R CODES

5.2.1 Edge Detection and Morphological Techniques:

```
## ---------- Required Libraries----------------------------------------

library(biOps)

library(png)

library(raster)

##------------------------------------------------------------------------


## ---------- Set working directory----------------------------------

setwd("C:\\Users\\Deepak Kumar Sisodia\\Desktop\\R- PROJECT")
```

```
##-------------------------------------------------------------------------

##------------------ function : pixelErosion()-----------------------------

# Input Parameters : a numeric (n X m) matrix containing 0s and 1s

# Return Value : a numeric (n x m) matrix containing 0s and 1s

# Description : Erodes the pixels in the image by-

# (1) Check all the surrounding 8 pixels for each foreground black pixel.

# (2) Convert the foreground black pixel (0) to background white (1) if black pixel is surrounded by atleast one white pixel.

#-------------------------------------------------------------------------

pixelErosion=function(imageinp){

 imagecopy=imageinp

 r=dim(imageinp)[1]

 c=dim(imageinp)[2]


 for(i in 1:r){


  for(j in 1:c){


   if (imageinp[i,j]==0){


    if(i!=r & j!=c){

     sub_image=rbind(imageinp[i-1,(j-1):(j+1)],imageinp[i,(j-1):(j+1)],imageinp[i+1,(j-1):(j+1)])

    }

    else{

     if(i==r & j==c)

     {

      sub_image=rbind(imageinp[i-1,(j-1):(j)],imageinp[i,(j-1):(j)])

     }
```

```
        else if(j==c)
        {
          sub_image=rbind(imageinp[i-1,(j-1):(j)],imageinp[i,(j-1):(j)],imageinp[i+1,(j-1):(j)])
        }
        else{
          sub_image=rbind(imageinp[i-1,(j-1):(j+1)],imageinp[i,(j-1):(j+1)])
        }
       }

       mask=matrix(0,nrow(sub_image),ncol(sub_image))
       xor.output=xor(sub_image,mask)

       if(sum(xor.output)>0){
         imagecopy[i,j]=1
       }

     }#close of pixel ==0
   }
 }

 return(imagecopy)
}


##------------------ function : pixelDilation()-----------------------------
# Input Parameters : a numeric (n X m) matrix containing 0s and 1s
# Return Value : a numeric (n x m) matrix containing 0s and 1s
# Description : Dilates the pixels in the image by-
# (1) Check all the surrounding 8 pixels for each background white pixel.
```

(2) Convert the background white pixel (1) to foreground black (0) if white pixel is surrounded by atleast one black pixel.

```
#-----------------------------------------------------------------------

pixelDilation=function(imageinp){

 imagecopy=imageinp

 r=dim(imageinp)[1]

 c=dim(imageinp)[2]


 for(i in (1:r)){

  #print(paste("i",i))

  for(j in (1:c)){

   #print(paste("j",j))


   if (imageinp[i,j]==1){


     if(i!=r & j!=c){

       sub_image=rbind(imageinp[i-1,(j-1):(j+1)],imageinp[i,(j-1):(j+1)],imageinp[i+1,(j-1):(j+1)])

     }

     else{

       if(i==r & j==c)

       {

         sub_image=rbind(imageinp[i-1,(j-1):(j)],imageinp[i,(j-1):(j)])

       }

       else if(j==c)

       {

         sub_image=rbind(imageinp[i-1,(j-1):(j)],imageinp[i,(j-1):(j)],imageinp[i+1,(j-1):(j)])

       }

       else{

         sub_image=rbind(imageinp[i-1,(j-1):(j+1)],imageinp[i,(j-1):(j+1)])
```

```
        }
      }

      mask=matrix(1,nrow(sub_image),ncol(sub_image))
      xor.output=xor(sub_image,mask)

      if(sum(xor.output)>0){
        imagecopy[i,j]=0
      }

    }#close if pixel == 1

  }
 }
 return(imagecopy)
}

##------------------ function : EdgeDetectionandImageMorphology()-----------------------------
# Input Parameters : a numeric (n X m) matrix representing a grayscale image
# Return Value : a numeric (n*m X 3) matrix
# Description : Creates and returns 3 features by doing edgedetection,
edgedetection+PixelDilation+PixelErosion, and
# edgedetection+PixelDilation+PixelErosion+PixelErosion+PixelDilation operations on the input image.The
# 3 features are combined and returned as a matrix having 3 columns with name -
"Edge","Edge+D+E","Edge+D+E+E+D"
# and n*m rows.
#-------------------------------------------------------------------------------------------

EdgeDetectionandImageMorphology = function(image.matrix){
```

```
  imagedata.inputimage=imagedata(image.matrix*255)

  imagedata.edgedetected = imgCanny(imagedata.inputimage, 0.7)

matrix.edgedetected=imagedata.edgedetected[1:nrow(imagedata.edgedetected),1:ncol(imagedata.edgedetected)]
  matrix.edgedetected.binary=matrix.edgedetected/255
  feature2= matrix(matrix.edgedetected.binary,ncol = 1)

  dilatedimage=pixelDilation(matrix.edgedetected.binary)
  erodedimage=pixelErosion(dilatedimage)
  feature3= matrix(erodedimage,ncol = 1)

  erodedimage2=pixelErosion(erodedimage)
  dilatedimage2=pixelDilation(erodedimage2)
  feature4= matrix(dilatedimage2,ncol = 1)

  feature.matrix=cbind(feature2,feature3,feature4)
  colnames(feature.matrix)=c("Edge","Edge+D+E","Edge+D+E+E+D")

  return(feature.matrix)
}

#read input image
inp.file=readPNG(".\\train1\\74.png")
#call function EdgeDetectionandImageMorphology() and pass input image matrix
# function will return a matrix with three columns and 1-row/pixel
```

```
# These 3 columns wil be 3 input features to the predictive model
output.feature.matrix=EdgeDetectionandImageMorphology(inp.file)
```

5.2.2 Median Filtering

```
library(png)
library(raster)



img = readPNG("C:/Users/mirak/Documents/Classes_Fall_16/R/Project 1/train/62.png")
CleanImg = readPNG("C:/Users/mirak/Documents/Classes_Fall_16/R/Project 1/train_cleaned/62.png")



plot(raster(img),col = gray.colors(256,start=0,end=1,gamma=2.2,alpha=NULL))
plot(raster(CleanImg),col = gray.colors(256,start=0,end=1,gamma=2.2,alpha=NULL))

 k = 9

 n = floor(k/2)

 #Modify the input image matrix by padding 0s outside the input image matrix
 #to make it size having an additional n rows and n columns
 img_modify = matrix(0,nrow(img)+2*n,ncol(img)+2*n)

 #Create an empty output matrix of the same size as the input image
 background = matrix(0,nrow(img),ncol(img))

 #Copy the Original matrix to the zero/padded matrix
 nrow(img)
```

```
ncol(img)
row_seq = seq(nrow(img))
col_seq = seq(ncol(img))

for (x in row_seq)
{
  for (y in col_seq)
  {
    img_modify[x+n,y+n] = img[x,y]
  }
}

View(img_modify)

#The new modified image matrix has additional 0s padded in the additional n rows and n columns
#surrounding the input image matrix
#View(img_modify)

#Store a k-k neighbour values in the array
#Sort and Find the middle element

for (i in seq(nrow(img_modify)-k))
{
  for (j in seq(ncol(img_modify)-k))
  {
    window = matrix(0,k*k,1)
    c = 1
    for (x in seq(k))
    {
```

```
    for (y in seq(k))
    {
      window[c] = img_modify[i+x-1,j+y-1]
      c = c + 1
    }
   }

   med = sort(window)

   background[i,j] = med[((k*k)+1)/2]
  }
}

plot(raster(background),col = gray.colors(256,start=0,end=1,gamma=2.2,alpha=NULL))

#the cleaned image can be obtained by substracting the original image from the background

foreground = img - background

#In this case, we know that the writing is always darker than the background,
#so, our foreground should only show pixels that are darker than background
foreground[foreground > 0] = 0

#Normalizing the final results (pixels) to lie between 0-1

m1 = min(foreground)
m2 = max(foreground)

foreground = (foreground - m1) / (m2 - m1)
```

```
 #Plot the raw-image, the background and the final cleansed image

 plot(raster(img),col = gray.colors(256,start=0,end=1,gamma=2.2,alpha=NULL))
 plot(raster(background),col = gray.colors(256,start=0,end=1,gamma=2.2,alpha=NULL))
 plot(raster(foreground),col = gray.colors(256,start=0,end=1,gamma=2.2,alpha=NULL))

 #Calculate the RMSE

 rmse = sqrt(mean((foreground - CleanImg)^2))
 print(c(rmse,k))

#Output the cleansed image

writePNG(foreground,"Image_Cleaned.png")
writePNG(background,"background.png")
```

5.2.3 Adaptive Thresholding

```
#load package
library(EBImage)

#Function to perform adaptive thresholding
Adaptive_thresholding <- function(image)
{

 # function to convert image matrx to vector
 img2vector = function(img)
 {
```

```
  return (matrix(img, nrow(img) * ncol(img), 1))
}

# function to convert image to matrix
image2Matrix = function(Img)
{
  m1 = t(matrix(Img, nrow(Img), ncol(Img)))
  return(m1)
}

#load the image
load_image = readImage(image)

#Create images with different height and widht
threshold_image_2 = thresh(load_image, 2, 2)
threshold_image_7 = thresh(load_image, 7, 7)
threshold_image_9 = thresh(load_image, 9, 9)
threshold_image_12 = thresh(load_image, 12, 12)
threshold_image_15 = thresh(load_image, 15, 15)

# combine the thresholded images
combine = cbind(img2vec(Image2Mat(threshold_image_2)),
            img2vec(Image2Mat(threshold_image_7)),
            img2vec(Image2Mat(threshold_image_9)),
            img2vec(Image2Mat(threshold_image_12)),
            img2vec(Image2Mat(threshold_image_15))
            )
#get the maximum pixel brightness among different images
final_image = apply(combine, 1, max)
```

```
 #form a final matrix
 image_matrix = matrix(final_image, nrow(img), ncol(img))
 #return the image
 return(image_matrix)

}
#Adaptive_thresholding("E:\\R Ram Gopal 2\\train\\2.png")
```

5.2.4 Feature Engineering

```
library(png)
# Loading the image
dirtyFolder =
"C:/Users/prana/Documents/Docs_2016/Uconn/Fall_2016/Assignments/Data_Analytics_using_R/Project/Project-1/Denoising_Dirty_Documents/Data/sample_test"
f = "1.png"
imgX = readPNG(file.path(dirtyFolder, f))

# Computing the row wise mean-pixel value for each pixel row in the image
vectorImgX = unlist(apply(imgX, 1, mean)) # Apply function will calculate row-wise "mean" pixel brightness for all the 258 rows in imgX matrix. The '1' parameter means apply the mean operation row wise.

# Plotting the mean pixel for each row versus the rows in the image
x = 1:nrow(imgX)    # There are 258 rows in imgX matrix.
plot(x, vectorImgX, col="black", type="l", xlab = "row", ylab = "mean brightness")

#### CLUSTERING BASED ON AVERAGE ROW BRIGHTNESS TO IDENTIFY THE LOCATION OF LINES OF TEXT:

# Making 2 cluster for brightness values: close to 0 (dark) and close to 1 (bright).
```

```
clstr = kmeans(vectorImgX, 2)


#Defining cutoff point as the mean of the highest value in the lower cluster and the lowest value in the lower cluster.

cutoff = ((max(vectorImgX[clstr$cluster == which.min(clstr$centers)]) + min(vectorImgX[clstr$cluster == which.max(clstr$centers)])) / 2)


# Marking a cutoff line on the 'mean brightness' versus row graph.

lines(vectorImgX * 0 + cutoff, col="red")


# Identifying all the rows where there a transition from white pixel (pixel value above the cutoff) to the dark pixel (pixel value below the cutoff):

spikes = c()

for(i in 2:length(vectorImgX))

{

  if(vectorImgX[i] < cutoff & vectorImgX[i-1] > cutoff)

  {

    spikes <- c(spikes, i)

  }

}


# Calculating the mean value of rows after which we expect line of sentence (spike/transition from high pixel value to low pixel value) to occur.

cycleLength = mean(diff(spikes))


# Creating a vector to find the spaces between the lines and initializing it with 0.

spaces = matrix(0, length(spikes) - 1)


# Creating a dummy vector (dummyImgX) to find the pattern of the transition of pixels in subject image for comparison later.
```

```
dummyImgX = vectorImgX


# Assigning all values which are greater than cutoff to pixel value of 1 (background pixel / brightest) and which are not making any changes to the values which are lower than cutoff.

for (i in 2:length(spikes))

{

  cycleStarts = spikes[i]                # In my case, spikes[1] is 15, i.e at row 15 of the pixel matrix, there was a transition from white to black.

  while (vectorImgX[cycleStarts] < cutoff) # While the value of "vectorImgX" (Mean value of the pixels in a given row) is less than that of cutoff.

    cycleStarts = cycleStarts - 1          # If the pixel is less than cutoff (i.e. dark), the value of cycleStarts is reduced by 1 to recheck the condition. WE WANT TO CHECK WHEN WE CAN EXPECT THE VALUE OF PIXEL THAT IS GREATER THAN CUTOFF.

  gapStarts = cycleStarts - 1              # This happens if the above while loop fails. At this point, gapstarts is also assigned a value, which is equal to the new (reduced) value of cycleStarts.

  while (vectorImgX[gapStarts] > cutoff)   # This loop will be executed when that pixel value will be greater than cutoff. At this point this while loop will execute to FIND THE GAP WHERE PIXELS ARE BRIGHT (greater than cutoff).

  {

    gapStarts = gapStarts - 1              # Value of gapstarts is reduced in every iteration to check the condition (in while loop) if the pixel value at that point is greater than the cutoff.

    dummyImgX[gapStarts]=1                 # At this point, we are assigning the pixel value of dummy vector equal to 1 (for comparison later).

  }

}



plot(x, dummyImgX, col="red", type="l", xlab = "row", ylab = "brightness")


# Initializing a new matrix to take the store the cleaned image. Initializing it to the original image and later compared it to the imputed value of background pixel.

imgCleaned = imgX

# Converting the new matrix into vector
```

```
matCln = matrix(imgCleaned, nrow(imgX) * ncol(imgX), 1)

#Imputing the detected gaps with a pixel value of 1.

matCln = ifelse(pmax(matCln,dummyImgX)>0.99,1,matCln)




#dumimg = matrix(dummyImgX, nrow(imgX), ncol(imgX))

#writePNG(dumimg,
"C:/Users/prana/Documents/Docs_2016/Uconn/Fall_2016/Assignments/Data_Analytics_using_R/Project/Project-1/Denoising_Dirty_Documents/Data/sample_test/sample1.png")


#final_image = matrix(matCln, nrow(imgX), ncol(imgX))

#writePNG(final_image,
"C:/Users/prana/Documents/Docs_2016/Uconn/Fall_2016/Assignments/Data_Analytics_using_R/Project/Project-1/Denoising_Dirty_Documents/Data/sample_test/sample2.png")


#Clearing all objects

rm(list = ls())


#Setting libraries

library(png)

library(raster)

library(xgboost)

library(biOps)

library(EBImage)

#_____
_____


start_time<-Sys.time()
```

```
#Get the input images

#Set folder where the images are located
train_folder<-"C:/Users/chars/Desktop/R/Project/R_Project/Data/train_sample"
target_folder<-"C:/Users/chars/Desktop/R/Project/R_Project/Data/clean_sample"

#Obtain a list of all the images in the folder
filenames = list.files(train_folder, pattern="*.png")

#_____
_____

#Load functions from the previous image cleaning techniques
source(file = "ImageCleaning.R")
#_____
_____

#Creating a dataframe that consists of the cleaned images from the previousimage cleaning techniques.
#These images are converted into a single column matrix and used as predictors for the XGBoost matrix

#Create am empty data frame. The clean images would be appended to this data frame
imgPred<-data.frame()



for(file in filenames){
  img = readPNG(file.path(train_folder,file)) #Load image
  clean_img<-readPNG(file.path(target_folder,file)) #Load cleaned image
  x1<-img2vec(img) #Convert image into a single column matrix. This would be the first predictor
```

```
  x2<-getLineWidth(img) #Obtain the image after performing featurization to extract line width.
                        #The output of the function is converted into a vector. This would be the second predictor
  x3<-img2vec(median_filter(img,9)) #Obtain the image after performing median filtering.
                                    #The result is converted into a single column matrix. This would be the third predictor

  #x4<-EdgeDetectionandImageMorphology(img)
  #x5<-img2vec(Adaptive_thresholding(file.path(train_folder,filenames[1])))
  y<-img2vec(clean_img)  #Convert the cleaned image into a single column matrix
                         #This would be the third predictor

  imgPred<-rbind(imgPred,cbind(x1,x2,x3,y)) #Combine all the predictors into a dataframe

}



#Gving column names to the dataframe
colnames(imgPred)<-c("Original","Featurization","Median_Filtering","Target")

#To train the model, we are selecting 250000 rows from the data frame
randSample<-sample(nrow(imgPred),250000)

#Convert the data into a dense matrix suitable to be the input for the xgboost model.
#The data is the matrix form of the first three columns of the dataframe and the label is the last column of the dataframe
train<-xgb.DMatrix(as.matrix(imgPred[randSample,-4]),label=imgPred[randSample,4])

#Running a cross validation on the dataset to find the optimum number of rounds to run the model. A maximum of 500 rounds
#is tried. The round with the minimum RMSE is taken as the number of rounds parameter for the model
```

```
cv_results = xgb.cv(data = train, nthread = 8, eval_metric = "rmse", nrounds = 500,showsd = T, early.stop.round = 50, nfold = 5,  print.every.n = 10)

#Getting the minimum number of iterations required to get the lowest testing data rmse

min_error_run = which.min(cv_results[, test.rmse.mean])



#Creating the xgboost model

xgb<-xgboost(data=train,nthread=2,nrounds = min_error_run)

end.time <- Sys.time()

time.taken <- end.time - start_time

time.taken

#_____
_____

#Testing the model with a sample image

#Reading image

pred_Img<-readPNG(file.path(train_folder,filenames[2]))

#Getting all the predictor values for the image using the image cleaning functions

x1<-img2vec(pred_Img)

x2<-getLineWidth(pred_Img)

x3<-img2vec(median_filter(pred_Img,9))

#x4<-mean_row_value(pred_Img)

# x5<-EdgeDetectionandImageMorphology(pred_Img)

# x6<-Adaptive_thresholding(file.path(train_folder,filenames[2]))
```

```
#Combining all the predictor values
pred_data<-cbind(x1,x2,x3)
#Naming the columns of the test data
colnames(pred_data)<-c("Original","Featurization","Median Filtering")

#Predicting the target values for the test data using the xgb model
pred<-predict(xgb,pred_data)
#Taking care of values outside [0,1]
pred[pred<0]<-0
pred[pred>1]<-1
#Converting vector back into a matrix of the same dimensions as the original image
clean_img<-matrix(pred,nrow(pred_Img),ncol(pred_Img))
plot(raster(clean_img),col=gray.colors(256,start=0,end=1,gamma=.85,alpha = NULL))
plot(raster(pred_Img),col=gray.colors(256,start=0,end=1,gamma=0.85,alpha = NULL))


#Estimating feature importance
imp<-xgb.importance(feature_names = colnames(pred_data),model = xgb)
xgb.plot.importance(imp)


#Writing the cleaned image to file
writePNG(clean_img,"clean.png")
```

5.3 CONSOLIDATED FUNCTIONS FILE

The consolidated functions file, "ImageCleaning.R" is present in the project folder.

6 REFERENCES

1. http://blog.kaggle.com/2015/12/04/image-processing-machine-learning-in-r-denoising-dirty-documents-tutorial-series/
2. https://arxiv.org/pdf/1406.1717.pdf
3. https://en.wikipedia.org/wiki/Median_filter
4. http://embeddedgurus.com/stack-overflow/2010/10/median-filtering/
5. http://stackoverflow.com/questions/11482529/median-filter-super-efficient-implementation
6. http://ceur-ws.org/Vol-1543/p1.pdf
7. http://angeljohnsy.blogspot.com/2011/03/2d-median-filtering-for-salt-and-pepper.html
8. https://colinpriest.com/2015/08/28/denoising-dirty-documents-part-5/
9. https://www.fonts.com/content/learning/fontology/level-2/text-typography/line-spacing-for-text
10. https://colinpriest.com//?s=denoising&search=Go
11. http://xgboost.readthedocs.io/en/latest/model.html
12. https://www.r-bloggers.com/an-introduction-to-xgboost-r-package/
13. https://www.analyticsvidhya.com/blog/2016/01/xgboost-algorithm-easy-steps/