ECS 132 Final Project

Joseph Lewis, Aaron Kaloti, Sean Malloy

due: Friday, March 24, 2017, 11:59pm

Contents

1 Image Processing

1.1 getnonbound(imgobj)

It was expected that the getnonbound function written to organize the data from an image into a format viable for operating on image smoothing and noise reduction would have a very long runtime. This could be expected, from the nature of the function - it has to store each of the grayscale data points in the image into a vector, whose length will be the number of pixels in the image (minus the border pixels) and also store each neighboring pixel value into a matrix of the same length, with four times the number of total entries.

If approached naively, the function could take many minutes to run for even one small file. For large files or many numbers of files, it could take even longer - a dramatic issue if this denoising regression is to be applied to many images by the same user. But, by avoiding R loops, which have additional overhead due to the nature of R as an interpreted language, and instead using vectorization, the function does not need to run slowly.

Our resulting function, as provided in the appendix, runs with no noticeable lag for the sample image provided, and quite quickly (in a matter of no more than a few seconds per image) for large (HD: 1920 x 1080) images (also provided in the appendix). We describe exactly how we got the function running faster in the code discussion in the appendix. We feel that this running time is acceptable for the process, as even with large images the user has no significant waiting period for the function to finish.

1.2 Regressing on Neighboring Pixels

The estimated prediction function from regressing y on x as run through the $getnonbound(imgobj)$ function are given in the equation below:

$$y = -0.007 + 0.15183 \cdot N + 0.15107 \cdot S + 0.35867 \cdot E + 0.35866 \cdot W \qquad (1.1)$$

We see that for this image, two significant trends emerge based on the coefficients. The first trend is that the neighboring east and west pixel grey-scale values contribute more than twice as much to the prediction of the center value than the north and south neighbors do. We can expect this from the nature of the *LaLa Land* picture: it is landscape, so it has many more horizontal pixels than vertical ones; and its color patterns show clearly in horizontal stripes, aided by the coloration of the sky across the top and the road through the bottom. We can use the sizing of the image (768 x 433), having nearly twice as many horizontal pixels as vertical ones, to explain the majority of this difference. But, we notice that the image size is not more than twice as wide as it is tall, so the

extra emphasis from the horizontal pixels as predictors must come from the patterns within the image - the sky has distinct shades that hold in horizontal lines, the mountains provide a horizontal stripe of similar color, etc. These patterns help skew the image towards favoring horizontal neighboring predictors.

The second trend is that the north and south pixels have nearly the same coefficients as a pair, and the east and west pixels also have nearly the same coefficients. This makes sense - that the pixels in an image both above and below the target pixel will have an equal effect on that pixel's predicted value, and the pixels in an image both right and left of the target pixel will also have an equal effect on that pixel's predicted value.

1.3 Image Smoothing

We were inspired in the following analysis by an article written by Robert Fisher, Simon Perkins, Ashley Walker and Erik Wolfart [1].

In typical image smoothing, we would use a 'kernel' to predict the pixel values. A kernel is just a size and a shape within which to look at additional pixels surrounding some center one. In our case, we used a plus shaped kernel that looked at only the orthogonally adjacent neighbors to an interior pixel in the image, and used standard linear regression techniques to examine how well those four neighboring pixels - north, south, east, and west - predicted their center.

A standard kernel to use in image smoothing is a 3x3 square centered on the interested pixel. This kernel takes the average values among *all* 9 pixels in the square (including the center), weighting them equally, and uses that average to predict the value for the center pixel, and replace it. The result is an image that has smoothed lines and can be somewhat blurred, but reduces any noise in the image.

However, it can be difficult to remove uniform (0,1) noise across an image. This type of noise is usually referred to as 'salt and pepper' noise, as it looks like white, grey, and black specks sprinkled across the image's surface. In typical average-based image smoothing, the averages generated about a noisy pixel can be significantly changed by the pixel itself being so far from the values around it. It skews the average and the resulting smoothing only adjusts the noisy sections of image so much, leaving a blurred and still speckled image.

In our image-smoothing algorithm, we will apply our plus shaped kernel, which does not include the center pixel, to each interior pixel in a 'salt and pepper' noisy image. By using a linear regression model (essentially, a weighted-average smoothing model, with the weights determined by the linear regression coefficients) which allows the patterns in the image and the image's size to influence the strength of any neighboring pixel's influence on the center pixel in each

kernel, we reduce the effect of noise further than simple average-based image smoothing, but not entirely. However, we also run the risk of noise on a horizontal scale having greater effect than noise on a vertical one.

1.4 Three Image Smoothing Examples

Now we example the regression-based smoothing on the following three images and discuss its merits. We first had to apply random 'salt and pepper' noise to these images. Finding the proper amount of noise to add was a matter of trial and error, and the function we used to apply the noise can be seen and is analyzed in the appendix. We settled on populating one percent of the pixels (0.01 proportionately) with random noise, as it was easily apparent that noise had corrupted the image, but not so noisy that the original image could not be made out or that the denoise function would be ineffective. Then we applied our regression in an attempt to remove the noise.

1.5 Example 1 - LaLa Land

Figure 1: The original image.

Figure 2: The image with U(0,1) random noise.

Figure 3: After the denoising algorithm is applied.

Notice that for this small of an image, the denoise function can only do so much. The patterns which are apparent horizontally do help reduce any vertical noise, but we can still see the 'salt and pepper' noise and the image lines looked blurred. To the smoothing algorithm's credit, the image does look considerably less speckled when viewed from a distance than its noisy counterpart.

The regression function for the original image is given in equation (1.1). The regression function for the noisy image shown above is:

$$y = -0.004 + 0.18614 * N + 0.18567 * S + 0.31975 * E + 0.31971 * W \quad (1.2)$$

We can see that the impact of the noise is to reduce the horizontal patterns in the image effect on the regression coefficients, but not by much. It is the

nature of the type of noise that still makes the de-noised image not appear very "fixed".

1.6 Example 2 - Dungeons and Dragons Image (High Resolution)

Figure 4: The original image.

Figure 5: The image with U(0,1) random noise.

Figure 6: After the denoising algorithm is applied.

This image was selected because of its balance of black and white sections, as well as its high resolution. The image provided the smoothest and cleanest result of all our test images after applying the regression-based smoothing. We note that this likely stems from: the image's equal areas of light and dark patches, which cause the overall regression not to be skewed towards one side or the other; and primarily the image's size, which allow the smoothing algorithm to more accurately remove noise due to more accurate predictions (predicting based on much more data) and resolution scaling (our eyes can only perceive so much detail, so adding more pixels overall means that small alterations in a few pixels here and there are not as easily perceived).

The regression function for the original image is given by:

$$y = -0.00228 + 0.29579 \cdot N + 0.29549 \cdot S + 0.21076 \cdot E + 0.21075 \cdot W \quad (1.3)$$

We see that despite the image's dimensions, the coefficients are larger for the north and south pixels than the east and west pixels. This is because the image presents clear vertical patterns - the vertical masts, the coloring in the dragon's necks, and the rock pillar at the left. This gives us an understanding that the more prominent factor for some images will be the patterns within the image, not the image dimensions. This is especially true for larger images, where the number of pixels to regress on is so large that the patterns really have time to enforce themselves on the regression coefficients and even themselves out.

The regression function for the noisy image is given by:

$$y = 0.00261 + 0.25500 \cdot N + 0.25491 \cdot S + 0.23780 \cdot E + 0.23786 \cdot W \quad (1.4)$$

We notice that this image presents a nearly even-average rounding method due to the addition of random noise. Note that now the regression function is

very nearly performing the same function as a simple average-based smoothing algorithm. However, though the differences between the coefficients are small, giving each adjacent pixel nearly equal weight, it is significant that we are not just purely using averages. The coefficients are the most optimized for this photo, and doing so for each photo results in the best looking de-noised images. If we merely applied the equal-weight averages to every photo, it would not always have the cleanest looking result, because we would be weighting the contribution of certain pixels improperly. We note that the noise applied to this photo was enough to skew the coefficients away from the patterns established and discussed in the previous paragraph.

1.7 Example 3 - Blast from the Past

Figure 7: The original image.

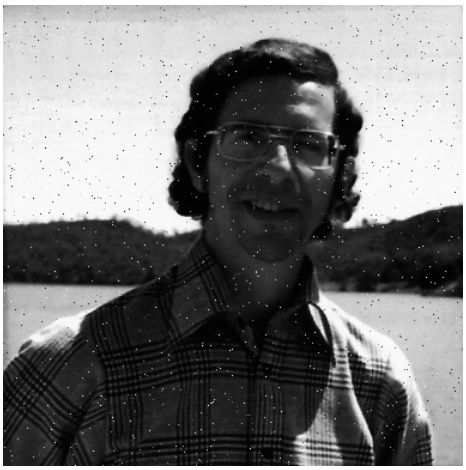

Figure 8: The image with U(0,1) random noise.

Figure 9: After the denoising algorithm is applied.

We chose this final image because of its square dimensions, to more closely examine the effect of dimension on the regression coefficients. We notice that there are definite horizontal patterns (the sky, the mountains in the background), so we assume that the coefficients will be skewed to favor the horizontal pixels, like in the *LaLa Land* picture. Let's examine (using **getCoeffs(imgname)** from the appendix):

The regression function for the original image is given by:

$$y = -0.00193 + 0.23920 \cdot N + 0.23914 \cdot S + 0.26273 \cdot E + 0.26322 \cdot W \quad (1.5)$$

We notice that the values of the coefficients are extremely close to one another, and close to being equally weighted in general, but we do have our expected skew towards the horizontal pixels. This enforces our idea that the patterns have some effect on the skew, because the dimensions of this image do not favor the horizontal or the vertical in terms of the number of available pixels. In this case, the vast majority of the patterns are broken because of the central figure, which splits the mountain range in the background and the sky with largely vertical patterns in his shirt and shadow, etc.

The regression function for the noisy image is given by:

$$y = -0.00006 + 0.24453 \cdot N + 0.24425 \cdot S + 0.25523 \cdot E + 0.25615 \cdot W \quad (1.6)$$

The random noise behaves as expected, moving the regression coefficients closer to an equal weight algorithm, as the noise acts to destroy any patterns present in the image and make the regression expect nearly equal contribution of each adjacent pixel towards the coloring of the center pixel in the kernal. This third case enforces our expectations for this pattern of equalizing the coefficients to occur. In essence, the randomly applied noise skews the averages so far from the original image that it will be impossible to fully restore it. But we can get close to the original image if we try to reduce this impact, which can be done by extending the kernal further, which is a common practice in image smoothing, as per the description at the beginning of this section.

2 Million Song Dataset

2.1 read.csv() vs fread()

On a speed comparison of **read.csv** compared to **fread**, **fread** is significantly faster to read data than read.csv is. The reason for this is that **fread** maps the file into memory and then iterates through the file using pointers, while read.csv reads the file into a buffer via a connection, as if it were a character, then attempts to convert the information into a integer or numeric values that could be used in a dataframe [2, 3]. The drawbacks to using **fread** is that it stores the file information into a data.table, which is unable to do certain functions that data.frame is able to do, since it is stored in a different format.

2.2 The Overloaded Operator of data.table

The author of **data.table** overloaded the subscript operator so that it returns a **data.table** object (instead of a vector like **data.frame**'s subscript operator does). The **length** function (for both **data.table** and **data.frame**) returns the number of columns, whereas the **length** function for a vector returns the number of elements. This explains why – in the example in the project specifications – **length(msdf[,1])** returns the number of rows while **length(msdt[,1])** returns 1.

According to [2], the author did this to have "consistency... when you use **data.table** in functions that accept varying inputs"; in other words, a function that takes a **data.table** could be more easily passed a specific column of that **data.table** without the programmer having "to remember to include **drop=FALSE** like ... in **data.frame**".

2.3 Difference of Means Analysis

Here we attempted to estimate population means for all songs using the sample means of the songs in our training set. The population includes all songs ever made, not just songs listed in the original data set, before we split into test and training. We first find estimates of the sample means.

$$\overline{X}_{pre1996V7} = -5.892 \tag{2.1}$$

$$\overline{X}_{post1996V7} = -5.459 \tag{2.2}$$

Then we can perform the two-sample T-test for difference of means, forming a 95% confidence interval, given as:

$$(-0.6134, -0.2522) \tag{2.3}$$

This tells us that the first population mean, for songs made before the year 1996, has a 95% probability to be between 0.6134 and 0.2522 smaller than the second population mean, for songs made during or after the year 1996. Be sure to understand that this probability arises from the fact that we are estimating population means of this data point for all songs from just the few songs we have in our training sets. Essentially, if we looked at all songs made in the year before 1996, we would be 95% certain that the mean of this data point would be between 0.6134 and 0.2522 less than the mean of the same data point for all songs made during or after the year 1996. We have this range because the sample means themselves are random variables, which depend on the random sample that we take, and thus we cannot be absolutely certain that their given means are precisely the population means.

The significance of this confidence interval is that 0 is not contained within it. Thus we prove the null (alternative) hypothesis wrong: there is a significant difference between the two population means (estimated by their sample means), and that difference lies somewhere on the interval, with the songs before 1996 having values more negative than those of songs during or after 1996.

2.4 Linear Regression Analysis

Our average squared prediction error of the V1 values in the test set is 90.16434. This error is acceptable, given that it is a *squared* error. Taking the square root of that error produces 9.49549 (meaning that, on average, our regression algorithm will be 9.49549 years off on its prediction of a song's year of release). This prediction error has a perhaps intuitive explanation: songs tend to change style and form by decade (each decade has a certain 'style' of music that we intuitively can recall), and a decade lasts 10 years (which is pretty close to our 9.49549). Thus, it makes sense that the difference between a song's year of release and the song's predicted year of release could tend to be a decade. That is actually quite a remarkable feat - each decade has its own style, and those styles are reflected enough in differences among the data given that we can then narrow the song back down to its decade just by the style as it shows through in audio data.

2.5 Logistic Regression Analysis

In this part, we classified songs by their data, with the classification categories being "before 1996" and "during or after 1996". Regarding the test set, of the songs we classified as coming out during or after 1996, 0.1898537 of them came out before 1996. In other words, based on our data sets, our regression method has a 18.99% chance of mistakenly classifying a song as having come out during or after 1996 when in fact it came out before 1996. This means that the introduction of auto-tune in the year 1996 has such a drastic affect on song's data that only being given the song's data forms, we can correctly predict the year of that song's release more than 80% of the time.

2.6 Principal Component Analysis

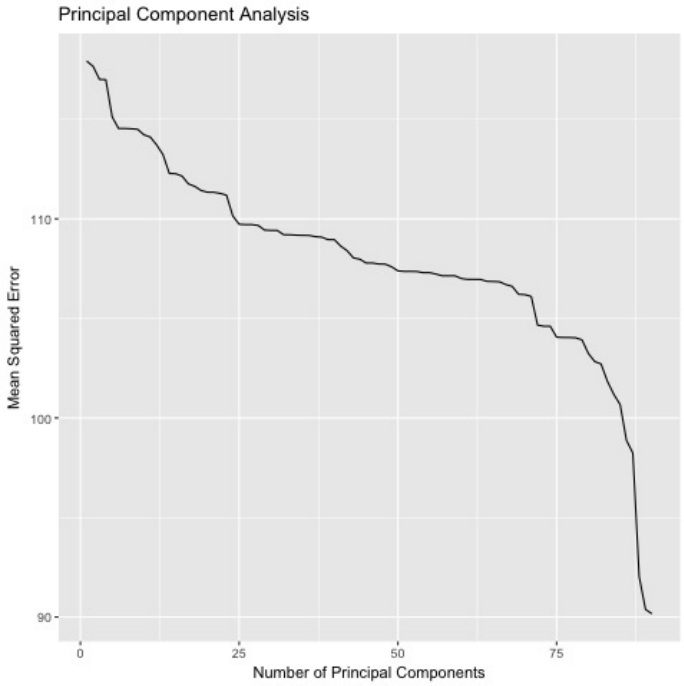

Figure 10: The graph of MSE against number of principal components included.

The above image shows the results of performing a version of principal component analysis on the million song data set. Here, we comprised the set of principal components for the data (the technical details of how that was accom-

plished are described in Appendix A.4) and used those components to form a new regression function. Then we performed the regression and calculated the errors as we continued to add principal components. We see that as we add more principal components to the analysis, the mean-squared error between the predicted years and the actual years for each song is reduced, until it comes down to the level of error that we saw in our original regression analysis. This is interesting behavior, because we expected to see the function behave slightly differently.

In regression analysis, there is an often arising problem known as "overfitting". We expect that after a certain number of components taking place in the regression, rather than reduce error as we continue to add new components, the error should actually increase. This is because once we have too many components, each one may be acting to influence our predictions in ways that the original components never did, and our prediction line is too responsive to the predictor variables. Then, as the predictor variables have expanded effects, the total prediction ends up far from the real value. The entire point of principal component analysis is to reduce the number of variables you use in your predictions until you have the optimal number to have the most accurate prediction function.

In our case, the error is continually reduced as we add more variables. Thus, we can make the conclusion that our prediction function is not "overfit". Since there are no more principal components to fit into our prediction function, then we know that the optimal prediction is given by using ALL of the principal components in the analysis, giving us the minimal error.

A Code

A.1 Problem A - The Code

```
# returns list of pixel values from imgobj
getnonbound <- function(imgobj)
{
  # grab the original image height and width from dimensions
  height <- dim(getChannels(imgobj))[1]
  width  <- dim(getChannels(imgobj))[2]

  # grab the matrix of all greyscale values
  data <- imgobj@grey

  # remove the first and last columns and rows (border pixels)

  # have to remove last 1st, so index doesn't change
  data <- data[,-width]
  data <- data[,-1]

  # same as before, remove last row first
  data <- data[-height,]
  data <- data[-1,]

  # create the y vector (all interior pixels stored as a vector)
  y <- as.vector(data)

  # make the X matrix (grabbing N / S / E / W neighboring pixels)
  x <- matrix(nrow = length(y), ncol = 4)

  entries <- 1:nrow(x)  # the row numbers of the X matrix
  xVals <- 2:(width-1)  # the default x coordinates for N / S
  yVals <- 2:(height-1) # the default y coordinates for E / W

  x[entries, 1] <- imgobj@grey[1:(height-2), xVals] # North Values
  x[entries, 2] <- imgobj@grey[3:(height-0), xVals] # South Values
  x[entries, 3] <- imgobj@grey[yVals,   1:(width-2)] # East  Values
  x[entries, 4] <- imgobj@grey[yVals,   3:(width-0)] # West  Values

  list(y = y, x = x) # the return list

  # Old Method for grabbing neighboring pixel values
  # Runs in O((height - 2) * (width - 2))
  # takes a lot of extra time due to R loops
  # entry <- 1 # start iterator for row count
  # for(xval in 2:(width-1)) # iterate over only interior pixels
  # {
  #   # iterate on y-coordinates second so that the format
  #   # exactly matches that of the y vector (column-major)
  #   for(yval in 2:(height-1))
  #   {
  #     # yval is listed first (column-major)
  #     nsewMatrix[entry, 1] <- greyMatrix[yval - 1, xval]
  #     nsewMatrix[entry, 2] <- greyMatrix[yval + 1, xval]
  #     nsewMatrix[entry, 3] <- greyMatrix[yval, xval + 1]
  #     nsewMatrix[entry, 4] <- greyMatrix[yval, xval - 1]
```

```
53    #     # move to the next row
54    #     entry <- entry + 1
55    #   }
56    # }
57  }
58
59  noise <- function(imgname, p)
60  {
61    require(pixmap) # need to read the image in as a pixmap object
62
63    image <- read.pnm(imgname)
64    height <- dim(getChannels(image))[1]
65    width  <- dim(getChannels(image))[2]
66
67    # get a count of the total pixels in the image
68    pixelCount <- height * width
69
70    # randomly generate p proportion to generate noise on
71    noisy <- sample(1:pixelCount, pixelCount * p, replace=FALSE)
72
73    # y coordinates are the remainder after we fit our pixel number
          ↪ into the image height evenly as many times as possible
74    yCoors <- (noisy %% height)
75    # if 0, then are exactly divisible and yCoordinate is the bottom
          ↪ edge (height)
76    yCoors <- ifelse(yCoors == 0, height, yCoors)
77
78    # x coordinates are the number of times we can fit our pixel
          ↪ number into the image height evenly (+1)
79    xCoors <- (noisy %/% height) + 1
80    # only occurs if the bottom right corner is a selected pixel, and
          ↪  dividing returns us exactly the width of the image
81    xCoors <- ifelse(xCoors > width, width, xCoors)
82
83    # convert x,y into a tuple and grab the corresponding index,
          ↪ replace each with random U(0, 1) values
84    image@grey[cbind(yCoors, xCoors)] <- runif(pixelCount * p, 0, 1)
85
86    write.pnm(image, paste("noised", imgname, sep="_"))
87  }
88
89  denoise <- function(imgname)
90  {
91    require(pixmap) # need to read the image in as a pixmap object
92
93    # grab basic image details
94    image <- read.pnm(imgname)
95    height <- dim(getChannels(image))[1]
96    width  <- dim(getChannels(image))[2]
97
98    # construct the regression coefficients using nearest neighbor
          ↪ prediction method
99    nonbound <- getnonbound(image)
100   coeffs <- lm(nonbound$y ~ nonbound$x, nonbound)$coefficients
101
102   # generate the predicted values (matrix multiply)
103   prediction <- as.matrix(cbind(1, nonbound$x)) %*% coeffs
```

```
104   prediction <- ifelse(prediction < 0, 0, prediction) # if values
          ↪ are below 0, set them to 0 (to avoid wrap-around)
105   prediction <- ifelse(prediction > 1, 1, prediction) # same for
          ↪ values above 1
106
107   interiorX <- 2:(width-1)  # the interior x-coordinates
108   interiorY <- 2:(height-1) # the interior y-coordinates
109
110   image@grey[interiorY, interiorX] <- prediction # apply the
          ↪ prediction values to all interior pixels
111
112   write.pnm(image, paste("de", imgname, sep=""))
113 }
114
115 tester <- function(imageNames, p)
116 {
117   for(name in imageNames)
118   {
119     noise(name, p)
120     denoise(paste("noised", name, sep="_"))
121   }
122 }
123
124 getCoeffs <- function(imgname)
125 {
126   require(pixmap) # need to read the image in as a pixmap object
127
128   # grab basic image details
129   image <- read.pnm(imgname)
130
131   # construct the regression coefficients using nearest neighbor
          ↪ prediction method
132   nonbound <- getnonbound(image)
133   coeffs <- lm(nonbound$y ~ nonbound$x, nonbound)$coefficients
134   print(coeffs)
135 }
```

A.2 Problem A - Code Discussion

The single largest obstacle to overcome with this program was the runtime. The process itself behind each function is rather simple: for **getnonbound** simply finding the greyscale values that correspond to the correct locations, for **noise** simply generating random U(0,1) noise across an image, and for **denoise** to run **getnonbound** and replace the interior pixels by predicted values using the regression of y on x. Each presented unique challenges in vectorization that would prove to significantly reduce the running time of the suite as a whole.

For **getnonbound**, the largest bottleneck on time was constructing the x matrix. At its core, it is a simple iteration problem, as shown in the commented out code for that function. We simply make a count of the current pixel we are trying to get values for, and then go through each pixel in the image by iterating

on x and y and getting the neighboring N / S / E / W pixels. We store this as a matrix and are done. But, the for loops in R cause significant overhead that can be reduced by vectorizing the arguments as we check pixel locations. By creating vectors of pixels to examine, we can relate the North pixels to the same x-coordinate and the (y-coordinate-1), which is shown through line 31, where the north pixels are all set at the same time, vectorizing access to the greyscale matrix. The same process is repeated for the South, East, and West pixel locations, and we can construct the final matrix without using any loops. This *significantly* reduces the runtime of this function, from several or more seconds for a small image (the *LaLa Land* image provided) to milliseconds, and from minutes for large images to only a few seconds. This does take more memory space though, as we have to create multiple large vectors of simple integer values across the entire width and height of the image.

For **noise**, the single largest bottleneck on time was applying the random noise to the grayscale matrix of the image. Generating the random pixels to apply that noise to could also have taken some time if done naively (without the proper use of the sample function or similar processes). Here we reduce that time bottleneck by vectorizing the X and Y coordinate pairs and binding them together in columns. This allows us to access each pair as a unit in the matrix and replace it with random noise, all at once as a vector. As written, the most significant reduction to the running time of this program are the two coordinate checks which validate any incorrect coordinate assignments due to mathematical processes (the ifelse statements). In addition, setting a large proportion of pixels to be noised in the parameter significantly increases runtime, because the sample must generate many more pixels and the vector arguments for the grayscale matrix and its x- and y-coordinates get much larger. These large vectors are again the cost of space that we trade off for faster running time.

For **denoise**, ignoring the runtime of **getnonbound** the largest time bottleneck was the application of the prediction values to each interior pixel in the image, sped up as in **getnonbound**'s construction of the x matrix. It also takes some time to process large vector arguments in the construction of the prediction values. Both of these are reduced by ensuring everything is vectorized, at the cost of memory space.

A.3 Problem B - The Code

```
# wrap all problem B into one function
probB <- function()
{
  data <- getTrainingAndTest(getData())
  meanAnalyze(data$training, data$test)
  print(regressionAnalyze(data$training, data$test))
  print(logisticAnalyze(data$training, data$test))
  principalAnalyze(data$training, data$test)
}

# reads the data in from the .txt file
getData <- function()
{
  require(data.table)

  #We use fread to read in the file "year Prediction MSD.txt" since it is significantly faster than read.csv
  #Using fread however does put our data into a datatable instead of a data.frame, limiting us on what functions we can run on our data
  fread("YearPredictionMSD.txt")
}

# get the training and test data sets from all data
getTrainingAndTest <- function(data)
{
  #Now we need to split our data into two parts: training set, and test set
  #We need training set to hold 2/3rds of the data, and test to hold the rest
  set.seed(101) # just a random seed to ensure we get random data
  sample <- sample.int(nrow(data), floor(2/3*nrow(data)), replace = F)
  training <- data[sample, ]
  test <- data[-sample, ]

  list(training = training, test = test)
}

#COMPARISON OF MEANS
meanAnalyze <- function(training, test)
{
  #We now need to split the training set into two parts
  #one part with the year of the data to be before 1996, and then the rest
  L <- (training$V1 < 1996) # a logical index vector

  trainingSplitPre1996 <- training[L,]   # all training data before 1996 (pre- autotune)
  trainingSplitPost1996 <- training[!L,] # all training data after 1996 (post-autotune)

  # we need to compare the value of v77 in both pre-1996 and post-1996 via means
```

```
45    trainingSplitMeanComparisonResults <- t.test(trainingSplitPre1996
          ↪ [,77], trainingSplitPost1996[,77])
46    print(trainingSplitMeanComparisonResults)
47  }
48
49  # find the mean square error for using ALL variables in the
        ↪ training set
50  regressionAnalyze <- function(training, test)
51  {
52    coeffs <- lm(V1 ~ . , training)$coefficients
53    computeAverageSquaredPredictionError(test, coeffs)
54  }
55
56  logisticAnalyze <- function(training, test)
57  {
58    # Fit logistic model
59    newFirstCol <- ifelse(training[,1] >= 1996, 1, 0)
60    g <- glm(newFirstCol ~ ., data=training[,-1], family=binomial)
61
62    # Now, we can find the predicted values, remembering that
63    # we're dealing with a logistic function
64    betasSum <- as.matrix(cbind(1,test[,2:ncol(test)])) %*% g$
          ↪ coefficients
65    predictions <- 1 / (1 + exp(-1 * betasSum))
66
67    # Find proportion of incorrectly predicted cases. More
          ↪ specifically,
68    # among all rows we predicted would have a year of at least 1996,
69    # find the proportion that are before 1996.
70    predictedClassifications <- (predictions > 0.5)    # True/false
          ↪ values
71    mean(as.matrix(test)[predictedClassifications,1] < 1996)
72  }
73
74  principalAnalyze <- function(training, test)
75  {
76    require(ggplot2)
77
78    P <- prcomp(training[,-1])$rotation # principal components matrix
79
80    # get the components from the training matrix and test matrix
81    components <- as.matrix(training[,2:ncol(training)]) %*% P
82    testComponents <- as.matrix(test[,2:ncol(test)]) %*% P
83
84    # so the computeAvSqPrErr function works
85    testComponents <- cbind(test[,1], testComponents)
86
87    # establish first colum of data as the song years
88    data <- as.data.frame(components)
89    errors <- c() # to store errors as we process them
90
91    # this takes a really long time to run, because it is not
          ↪ optimized
92    # but I am unsure of how to optimize it any better.  We have to
          ↪ do the lm 90 times
93    for(i in 1:ncol(components))
94    {
```

```
95      # find the mean square errors for each component
96      PredictorVariables <- paste("components[,", 1:i, "]", sep="")
97      Formula <- formula(paste("training$V1 ~ ", paste(
            ↪ PredictorVariables, collapse=" + ")))
98      coeffs <- lm(Formula, data)$coefficients
99      errors <- append(errors, computeAverageSquaredPredictionError(
            ↪ testComponents[,1:(i+1)], coeffs))
100   }
101
102   # now plot the results
103   plotData <- data.frame(components = 1:ncol(components),
          ↪ meanSquareErrors = errors)
104   plot <- ggplot(plotData, aes(x=components, y=meanSquareErrors)) +
          ↪ geom_line()
105   plot + labs(title="Principal Component Analysis", x = "Number of
          ↪ Principal Components", y = "Mean Squared Error")
106 }
107
108 # returns the mean squared error between the test set and a test
      ↪ set
109 computeAverageSquaredPredictionError <- function(test, coeffs)
110 {
111   # grab the predicted values using matrix multiplication
112   predictions <- as.matrix(cbind(1, test[,-1])) %*% coeffs
113   # Then compute average squared prediction error over all the test
          ↪ set.
114   meansSquared <- mean((predictions - test[,1])^2)
115 }
```

A.4 Problem B - Code Discussion

Our main difficulty in this problem was trying to get **PrincipalAnalyze** to work, because at the surface the task was quite easy but none of us wanted to type out 90 **lm** commands, even though it may have solved the problem faster than our workarounds. Eventually we settled on the slow but working code presented in the listing, and discussed further below. We also spent some time making sure the code was robust and making each function have a specific purpose, so that we could test the parts of this problem independently of one another.

For **getData**, we used **fread** to read in our "YearPredicionMSD.txt" file (the MillionSong Data Set), since it was significantly faster in its performance than read.csv. We discuss exactly why that is in section 2.1.

For **mean analyze**, we were able to compare our data for the training set of pre-1996 and post-1996 by using the Welch two sample t.test function for confidence testing, to find the means of the two V77 variables. The selection of this variable was arbitrary. The Welch two-sample t test gives us some results, including the sample means of each variable and a 95% confidence interval for

their difference. We discuss these implications in section 2.3. The trickiest part of this code was defining the logical indexer so that we could quickly and easily separate the training data into those songs before 1996 and those after.

Our final version of **regressionAnalyze** was simple, beginning with obtaining the estimated regression coefficients. We then used those coefficients to form the linear regression equation and "predicted" values for V1 in the test set. This prediction process was extremely tricky until we realized that multiplication of the test set matrix with the vector of regression coefficients would get us our predicted values for V1. Note that – in this process – we had to substitute a column of 1's for the (unneeded) V1 values in the test set, so that the matrix multiplication would work out.

Our final version of **logisticAnalyze** also turned out simple, beginning with finding the regression coefficients. Because this function used logistic regression, calculating the predictions (of whether each song was released before 1996 or not before 1996) used a different equation than linear regression did. Among the songs in the test set that we predicted were released during or after 1996, we calculated the proportion of them that came out before 1996 (thus meaning that they were erroneously classified). The main trick here was dealing with data.table's lack of support for using a vector of true or false values (**predictedClassifications** in the above code) to index a data.table object; we solved this problem by – in this function only – converting the test set data.table object into a matrix and *then* indexing it with the vector of true or false values.

The **principalAnalyze** function performs principal component analysis of a sense. We perform linear regression after finding the principal components to the song data. The principal components are given as linear combinations of the original V2, ..., V91 data information, so we have to multiply the P matrix through the original data to get a new matrix of the components, with a row for each song. Then we can perform the regression, each time adding the next most principal component until we are regressing on all components (this is handled in the for loop, which is not-optimal but was as best as we could get the code working for this project). The PredictorVariables variable stores a copy of strings of all of the columns in the components matrix that we need to perform our regression with, and it is collapsed into a formula to be read by the **lm()** function which allows us to include one additional variable each time we iterate through the for loop. Then we calculated the error from these predicted values by principal component using the same method as in the original regression analysis.

A.5 Plot Converter Function

As supplied by Norman Matloff in the project description, used to save the plotted graph from Problem B.

```
pr2file <- function (filename)
{
  origdev <- dev.cur()
  parts <- strsplit(filename,".",fixed=TRUE)
  nparts <- length(parts[[1]])
  suff <- parts[[1]][nparts]
  if (suff == "pdf") {
    pdf(filename)
  }
  else if (suff == "png") {
    png(filename)
  }
  else jpeg(filename)
  devnum <- dev.cur()
  dev.set(origdev)
  dev.copy(which = devnum)
  dev.set(devnum)
  dev.off()
  dev.set(origdev)
}
```

B Distribution of Labor

B.1 Joseph Lewis

R coding for Problem A, first and final drafts for the writeup of Problem A and related appendix section, research on image smoothing processes, management of Github repository for project, assistance on Problem B coding and testing for overall coding and discussion details on **principalAnalyze** function, assistance with optimization and vectorization of all coding, difference of means analysis and principal components analysis, final write-up review and submission.

B.2 Sean Malloy

Made contributions on the creation of Problem A, such as suggestions and code testing. Edited the term paper/made contributions of the creation of the paper. Made the rough draft for the coding for problem B, and related research on the topics for part B. Helped with Problem B code discussion. Final write-up review.

B.3 Aaron Kaloti

Helped work out the approach to Problem A's coding. Wrote certain parts (regarding problem B) of and proofread the term paper. Set up Bibtex. With help, wrote much of the code for the linear regression (**regressionAnalyze**) and logistic regression (**logisticAnalyze**). Final write-up review.

B.4 Luhong Pan

Thanks so much for the wonderful quarter! You were one of the best TAs we've ever had! :)

B.5 Norm Matloff

Plot converter function. Thanks for the awesome quarter, we couldn't have done it without you! :)

References

[1] Ashley Walker Robert Fisher, Simon Perkins and Erik Wolfart. Spatial Filters - Mean Filter. http://homepages.inf.ed.ac.uk/rbf/HIPR2/mean.htm.

[2] Frequently Asked Questions About data.table. https://cran.r-project.org/web/packages/data.table/vignettes/datatable-faq.htmlj-num.

[3] Reason Behind Speed of fread in data.table Package in R - Stack Overflow. http://stackoverflow.com/questions/24424361/reason-behind-speed-of-fread-in-data-table-package-in-r