# ECS132: Term Project

Chad Pickering, Chirag Kashyap

March 25, 2017

## 1 Introduction

The two applications of probability and statistics to computer science contained in this project are a fair representation of the skills and concepts learned in ECS132. For each problem, we state a premise of the question, establishing scope and utility, as well as present the details of each process, such as justifying programming decisions and statistical assumptions, and explaining computational procedures and analysis.

## 2 Problem A: An application of linear regression methods to image processing

### 2.1 Premise

**Multiple linear regression** is a critical and widely-used methodology in image filtering and processing, particularly where smoothing is concerned. Its practical use is not only highlighted in its straightforward mathematical nature, but also in its flexibility and compatibility with images of all sizes, colors, and attributes. In this application, we used multiple linear regression models to predict the interior pixel values of a corrupted black-and-white image, where a given randomly selected proportion of the pixels had been overwritten with uniformly distributed noise. To this end, our predicted values of each interior pixel creates a "smoother" resultant image, with the magnitude of smoothing dependent on the proportion of noisy pixels introduced.

### 2.2 Procedures

**Note:** *The R functions created for this application are generalized for a black-and-white image of any dimensions. However, for the sake of this problem, one particular image, referred to as the "original image" throughout the following analysis, was chosen for demonstration.*

### 2.2.1 Building the Neighbor Matrix with getnonbound()

The **original image** has dimensions of 433 by 768 pixels; each of these 433×768 pixels, then, is assigned a value in the range [0,1], darkest to lightest, representing its true greyscale "identity". The full pixel array is extracted using the R package **pixmap** for all pixels, and the original image can be displayed in full greyscale.

Figure 1: Original image.

The pixels with neighbors on all sides (e.g. not on the border of the image) will be referred to as **interior pixels** (whereas the pixels on the border will be dubbed **exterior pixels**). Because each of these interior pixels' greyscale values will eventually be predicted by the to-be-generated regression model, a vector of all 431·766 values is created, denoted as $\overline{Y}$. This vector contains all true interior pixel greyscale values by column, from left to right.

For each interior pixel, here, arbitrarily denoted $y_i$, four immediately neighboring pixels exist adjacent, denoted as follows in clockwise order starting from the pixel just above $y_i$, $x_i^N$, $x_i^E$, $x_i^S$, $x_i^W$, where the top pixel will be referred to as the northern pixel for the remainder of the procedure, and so on. These directional superscripts are simply for clarity, and never take on any numerical values. So, each interior pixel $y_i$ has its own unique set of four neighbors partnered to it.

We can then take the vector of all $x_i^N$ values in the same column-wise order as is $\overline{Y}$ and create another vector, $\overline{X^N}$. Analogous vectors for the southern, eastern, and western adjacent pixels are denoted as $\overline{X^S}$, $\overline{X^E}$, and $\overline{X^W}$, respectively. When all four vectors are placed into a matrix with row order matching that of the order of the vector $\overline{Y}$, the

dimensions are $431{\cdot}766 \times 4$, one row per interior pixel, and one column per neighboring pixel - we will call this the **neighbor matrix**.

### 2.2.2 Alternatives to Reducing Function Runtime by Vectorization: Pre-Allocating Matrix Dimensions

In building the **getnonbound()** function to generate the matrix, the runtime is greatly reduced when operations are vectorized. This way, R does not have to re-allocate memory with every iteration; this slows the operation down proportional to the size complexity of the object being operated on.

However, a more direct alternative of standard R vectorization with functionals (e.g. **sapply(), etc.**) does informally exist, and we have exploited this fact. Within the function, we directly pre-allocate memory (just as the source code of functionals do) using an empty matrix exactly the dimensions required before the data is loaded into it. This pre-allocation is exactly what vectorization requires (and why it is typically so much faster), but this method is extremely explicit. This method resulted in elapsed total runtimes of about 2.28 seconds on average. [6]

### 2.2.3 Understanding the Regression Function for the Original Image

When regressing $\overline{Y}$ on all four neighbor vectors, we get the following regression equation:

$$\hat{y_i} = \hat{\alpha} + \hat{\beta_N} x_i^N + \hat{\beta_S} x_i^S + \hat{\beta_E} x_i^E + \hat{\beta_W} x_i^W$$

When the original image is used (without any noise added), the equation is as follows (estimated coefficients rounded to four decimal places):

$$\hat{y_i} = -0.0112 + 0.1711 x_i^N + 0.1702 x_i^S + 0.3391 x_i^E + 0.3391 x_i^W$$

We first notice that the estimated coefficients for north and south are almost the same, as is for east and west. This can be interpreted as if either the north or south pixel's greyscale value increased by one unit (e.g. from 0 to 1) the increase of the estimated central pixel value would be about 0.17. In contrast, if either the east or west pixel's greyscale value increased by one unit (e.g. from 0 to 1) the increase of the estimated central pixel value would be about 0.339, approximately double the effect of the north and south pixels. When looking at the original image, this intuitively makes sense because there is more consistency in the image when traversing horizontal pixels (average row-wise variance of 0.0333) than with traversing vertically (average column-wise variance of 0.0409); there is more variance in any given column in contrast with any given row, on average. Thus, predicting a pixel's value from just its west and east neighbors is likely to be more accurate than with just its north and south neighbors.

### 2.2.4 Low-Pass Filters and Convolution Kernels

There exists many methodologies adjacent to regression in the area of image smoothing, and to emphasize a crucial point about why regression modeling works, it is beneficial to discuss the concept of using weighted averages. In a sense, the calculated estimate of a pixel's greyscale value is a weighted average of the surrounding pixels, with the four beta estimates as the respective weights in our example. In the paper 'An Adaptive Window Mechanism for Image Smoothing' by Goshtasby, et al., it is indicated that any smoothing techniques used on solitary images is done in the spatial domain rather than the temporal domain (through time), where smoothing reduces noise while preserving image structure. This smoothing technique is also known as a **low-pass filter**; it averages out abrupt deviations in magnitude in the image by calculating the average of each pixel and several surrounding neighbors, replacing the central pixel with the resulting value. [2] [4]

Sometimes, depending on the nature of the image, applying a low-pass filter can suppress noise that masked slight detail and gradual changes in the image. It is worth noting that pixel intensities are averaged in a square region regardless of image dimension or non-symmetry of the image. This procedure can be adjusted for dimension and symmetry, but for the purpose of our analysis, the square neighborhood (with the four neighboring pixels, as described) will remain.

If we were to choose a completely weighted methodology, our beta estimates would suffice (adjustment is needed - more discussion later); this choice can be visualized with what is known as a **convolution kernel**, an m×n grid that shows how each respective pixel is weighted to determine the value of the central pixel. For example, a low-pass filter applying equal weight to each adjacent/neighboring pixel would result in a grid with pixel weight $\frac{1}{mn}$ (assuming all in the region are nonzero). [2]

Our ambitions are similar, but we would have to adjust the beta estimates so that they sum to 1. This way, we can form a grid such that the kernel would have non-zero values only at the northern, southern, eastern, and western pixels, with values approximately equal to the estimates. Note that these estimates would vary as each simulation of random noise yields different results.

| | | |
|---|---|---|
| 0 | 0.1711 | 0 |
| 0.3391 | 0 | 0.3391 |
| 0 | 0.1702 | 0 |

| | | |
|---|---|---|
| 0 | 0.1678 | 0 |
| 0.3326 | 0 | 0.3326 |
| 0 | 0.1669 | 0 |

Figure 2: Convolution kernels (beta estimates [top], adjusted beta estimates [bottom]).

Again, this weighting methodology is an adjacent method, and its similarities are meant

to stimulate thought about what the function of the beta estimates essentially are. Our regression model was fit to minimize the sum of squared error, so we have to account for an intercept term and assumptions of linearity and normality that a simple weighting scheme does not have to account for.

### 2.2.5 Applying Noise and Employing Smoothing with denoise()

To demonstrate our smoothing mechanism based on the regression model, 20% of the pixels in the original image are randomly chosen to be corrupted - true pixel values are replaced by uniformly distributed random noise. This appears as a thin, yet immediately apparent layer of grey 'snow' over the entire image, as follows below. This noisy image is the input of the **getnonbound()** function to generate the neighbor matrix (recall Sec. 2.2.1).

Figure 3: Original image, corrupted by uniformly distributed random noise.

In the function **denoise()**, we call **getnonbound()** to generate both the vector $\overline{Y}$ and the neighbor matrix; these are needed to fit a regression function with unique values of $\hat{\beta_N}$, $\hat{\beta_S}$, $\hat{\beta_E}$, $\hat{\beta_W}$ specific to the noisy image generated. We then use matrix multiplication to multiply the values of the neighbor matrix by the vector of beta coefficients, which results in a vector $\overline{Y}_{pred}$ with the same dimensions as the vector $\overline{Y}$ containing the fitted values of what we will call the **predicted image**. Some of these values may be slightly larger than 1 or slightly smaller than 0, so the range is truncated to the interval [0, 1]. Recall that only interior pixels are predicted, so external pixels will remain noisy. This new vector $\overline{Y}_{pred}$ can then be plotted, and the predicted image can be seen. An example is below.

Figure 4: Predicted image.

The effects of the smoothing is quite apparent - by using the values of the four neighboring pixels, we can compute a predicted value of each interior pixel based on the unique multiple regression function. It is worth noticing that the image has less extreme black or white pixels (values close to 0 and 1, respectively) - the predicted image is made up of averages of pixels in small defined regions. The implication is that as the proportion of uniform noise increases, the more smooth the image gets; this is because the predicted values generated by the regression function are more likely to be calculated based on pixels with uniform noise (as they are more commonly found in the image), and the approximate mean of a large sample of U(0,1) values is about 0.5, which is moderate grey (neither extremely black nor white).

### 2.3 Conclusion and Further Applications

There are several possible variations of the methods and procedures used in this analysis, some of which are much more high level. But, it is reassuring to know that a fundamental applied statistical method such as multiple linear regression can be used in such a powerful way to reduce noise in an image. In a future analysis with a more ambitious application, we could regress our vector of interior pixel values on a matrix of columns such that the distance from the pixel of interest to some other neighboring pixel (represented by some column) is weighted by distance, similar to a Gaussian filter.

# 3 Problem B: Applications of regression and PCA on audio data to explore prediction and error

## 3.1 Premise

Now, we would like to expand our use of linear regression techniques to predicting the release year of songs based on several continuous variables containing audio data. By splitting the entire dataset into two distinct sets, we will be able to generate a linear regression function based on the training set and apply it to the test set to see how well prediction does overall. Additionally, logistic regression will be employed to predict whether a song was released before 1996 (corresponding to the introduction of autotuning) or not. Finally, we will regress year against the principal components of all of the audio variables in the training set and compute error in the test set. This process will show some very important results in terms of how mean squared error behaves as number of principal components used in a model increases.

## 3.2 Procedures

**Note:** *Our discussion in this section will follow the order requested. Please find these split into their respective sections below.*

### 3.2.1 Data Input, Storage, and Access

Before any analysis is done on the data, we need to consider the data itself and how it is read into R. There are two main ways to accomplish reading in very large text files, the first being with the function **read.csv()**, which took a total of about 5.8 minutes on the audio data. The second way is with **fread()** from the CRAN package **data.table**, which took a total of about 2.8 minutes. Why does **fread()** take significantly less time? What is the logic behind this discrepancy?

The **read.csv()** function reads all of the content into memory as a large character matrix as if every column were of type character, and then later tries to coerce appropriate columns to numeric types or factors in a following step - this takes quite a long time for datasets with hundreds of thousands or millions of rows. In contrast, the **fread()** function merely reads in each column as if the content were of type character; by avoiding the coercion step altogether, a majority of the time is saved. An additional reason why **fread()** is significantly faster is because the data that is read in is not physically copied in a system's memory; rather, the object created is just a copy of column pointers. This memory efficiency also makes computational speed a lot quicker. [3] [1]

When we compare common storage decisions, we encounter a big difference between the common **data.frame** and the **data.table** and how attributes of columns and subsets are accessed. Whereas in a **data.frame** we can access the contents of the first column with call form `df[,1]`, in a **data.table**, `dt[,1]` simply returns 1, and for the fourth column for instance, `dt[,4]`, returns a 4. Thus, when length is called on any of these `dt[,i]` calls,

it just returns 1 because the result of all of those aforementioned calls returned vectors of length 1.

Interestingly, this is a purposeful design choice by the author of this package. The second argument within the brackets is an expression to be evaluated within the scope of the **data.table** object because, it is claimed, it is bad practice to refer to columns by number because if the ordering is changed, the code will not be referring to that specific column by name and the result of the code will change completely. (By scope, we mean the environment where the names of the columns themselves are variables to be used.) This design choice increases clarity, as it is not immediately obvious to the writer of the code or those who read it later which column is the $i^{th}$ column, so referring to it by its explicit name (without quotation marks) is best. With this scheme, more flexibility is allowed - any R expression can be placed in the second parameter or wrapped with `list()` if appropriate. Instead of calling `length()` on a specific **data.table** column, we can instead use the `.N` parameter to return the number of rows in the column (or whatever subset requested), and the parameter `by=` to group rows in some fashion. [5]

### 3.2.2 The Training and Test Sets

The entire dataset has 515,345 rows, which is a very healthy sample from which to generate a training and a test set. The training set consists of two-thirds of the rows of the full dataset, randomly sampled; the remaining third forms the test set. These proportions are at a good ratio such that the training set is large enough to build strong models (linear and logistic regression are coming up in the discussion) and the test set is just large enough to understand how well those models can predict and how their error behaves.

### 3.2.3 Comparing Two Means of an Audio Variable

Say an audio variable is chosen at random, like V77, and we would like to compare the means of its values before Autotune was introduced and afterwards. Within the training set, we split the data into these two groups (pre-1996 and otherwise) and use a 2-sample t-test with $H_o : \mu_{before} = \mu_{after}$ as our null hypothesis and $H_A : \mu_{before} \neq \mu_{after}$ as our appropriate two-sided alternative hypothesis. We will test this with a type I error ($\alpha$) threshold of 0.05, and will generate the corresponding 95% confidence intervals as well. Our sample size is so high in both samples that our t-distribution essentially converges to an approximate Z-distribution.

Thus, when the test statistic of -3.6454 is compared to the critical value of about $Z_{0.025}$=-1.96, we can say that since the statistic is inside the rejection region (less than -1.96), the null hypothesis is rejected. The p-value here is 0.000267, which suggests that if the population means were truly equal in actuality, then we would have about a 0.0267% chance of observing our test statistic or one more extreme. Therefore, we have sufficient evidence to suggest that the population mean audio value for V77 before Autotune is significantly different from the population mean audio value for V77 after Autotune was introduced.

We can better confirm this with the corresponding 95% confidence interval of (-0.5176, -0.1556). A traditional interpretation would suggest that since 0 is not contained in this interval, we can conclude that we are 95% confident that the true population mean of V77 before Autotune was significantly different from the true population mean of V77 after Autotune; however, we have more information than that. We can see that both values are negative - this suggests that the population mean of V77 after Autotune was introduced is higher than the population mean of V77 pre-1996. But, we do have to be cautious. The upper bound of the confidence interval is quite close to 0, and the range is a bit wide compared to the scope of the difference, especially when factoring in the very high sample size. So, we have a fair amount of evidence that the true population means are different, and that the post-1996 population mean is likely higher than the other, but the entire landscape of the question has to be taken into account, which introduces at least a small amount of skepticism.

#### 3.2.4 Linear Regression and Mean Squared Prediction Error

Using the training set, we regressed the year column on all of the audio variables in order to predict, given new audio data points (from the test set), which year a song was recorded in. All of the audio variables look very similar in their distribution and ranges, but no details were given on how to interpret any of them. So we will just assume that they represent auditory indicators that could be used as discriminatory factors or significant clues as to which year the songs belong.

When the multiple linear regression function is fit based on the training set yielded from a particular run, at first glance a majority of the beta estimates **appear** to have very significant effects on the release year of a song because most p-values are very close to 0. But by looking at the output, we can also see that a majority of the betas are quite close to 0, and even if those are deemed significant predictors (in that they deviate from 0 significantly and therefore have an effect on year), their effect could be minuscule in comparison to the few variables that have larger beta estimates with mild to high significance. The point estimate of the intercept term is about 1951, which implies that if all audio variables were 0, then the predicted year would be 1951; however, since we do not know how to interpret the audio variables, we do not understand if the intercept term has a relevant meaning to us.

There are four variables, V2, V7, V9, and V12, that all have beta estimates farthest from 0 in comparison with the rest and have p-values near 0 - these four would likely be kept in a final model generated by traditional model selection procedures (e.g. forward selection with AIC, etc.). For example, with all other variables held constant, we can say that a one unit increase of $X_2$ (corresponds to V2) will result in an approximate 0.875 increase in the predicted year. This suggests that as the years went on, the numerical value of V2 tended to increase (perhaps Autotuning was a factor, but we certainly cannot conclude this whatsoever). Similar interpretations can be developed for the other variables, with the direction of effect on release year determined by the sign on the particular beta estimate.

The mean squared prediction error (MSPE) was computed to be about 90.89855 overall

on this full/saturated model. This can be interpreted as the expected value of the squared differences between the values fitted by the linear regression model and the observed values in the test set (what we tried to predict). MSPE is a method of estimating variation between model predictions and observed values, where low MSPEs indicate that models have strong predictive power. The MSPE of a model must be compared to MSPEs of other models to draw conclusions. For example, we know that adding a predictor cannot tell us less information about the response, but does this new predictor tell us a significant amount of information such that it is worth having an additional parameter in the model? Using this mentality, we can choose the best models using best subset selection (very computationally expensive) or, as it was suggested earlier, traditional model selection procedures. Typically, though, predictive models should include more predictors, so we would use backward selection (starting with the full model) with the AIC criterion, which does not penalize number of parameters as much as the BIC criterion.

### 3.2.5 Logistic Regression and Classification Error

Within the training set, we can further stratify the data based on if the songs were released before 1996 or not. From an outside perspective, the year 1996 seems like a very random year to differentiate between songs. However, 1996 was the year that Autotune was introduced and implemented in the music industry. The program is used to alter pitch in the vocal and musical aspects of songs, and its use became commonplace in several facets of the industry very quickly. Because of its rapid spread, we want to know if we can predict whether a song was released before 1996 or after using logistic regression on the audio variables used in previous sections. If the error rate on our predictions is low, we can conclude that Autotuning was likely a major factor in the numerical shifts in most if not all of the audio variables. If the error rate is high, or prediction is worse than a coin flip, we can say that Autotuning did not significantly change the audio variables too much or at all.

Since simple logistic regression requires a binary variable, we created a new column called 'autotune' in both the test and training data sets. We defined 1 to be songs created before 1996 and 2 to be songs created during or after 1996. This allowed the logistic regression to be run properly and our prediction results to make sense. After we ran the regression, we noted similarities between the linear and logistic regressions. Many of the same predictor variables that had high z-values in the logistic regression had high t-values in the linear regression. High (or significantly deviated) Z-values and t-values are indicators of if a variable is significant enough that it merits a place in the final, condensed model (through model selection criterion). In particular, V2, V3, V4, V7, and V14 all had |z-values| higher than 30. In our linear regression, those variables had the five highest |t-values|. This indicates that those 5 variables are very good predictors of year, not just if a song was released before 1996 or not.

After fitting the model, we predicted the test data set using response. This gave us a probability of whether or not the song was released before 1996 or not. We defined any probability over 0.5 to be True and then added one. This made any probability below 0.5 to be 1 and any probability over 0.5 to be 2, resulting in the binary variable we wanted. Next,

we used made a classification table to see how many songs were misclassified. We ended up with a reasonable overall error rate of 0.2132. This can be interpreted as the probability that a song is misclassified when predicted through this logistic regression model. From the following table, songs that were released previous to 1996 were more commonly misclassified as songs released after 1996, with a classification rate of 0.444. Songs that were released after 1996 had a much higher classification rate, right around 0.919.

| | model | |
|---|---|---|
| true | 1 | 2 |
| 1 | 21241 | 26595 |
| 2 | 10025 | 113921 |

Figure 5: Classification Table from Logistic Regression

While our logistic model does a great job of correctly predicting songs that were released after 1996, it lacks the power to correctly predict songs that were released before 1996. This could be because of confounding variables or a myriad of other reasons. In any case, the high classification rate of songs released after 1996 suggests that it could be possible that Autotune made it easier to predict if a song was released after 1996.

### 3.2.6 Principal Component Analysis and Mean Squared Error

Principal Component Analysis (PCA) is a method of reducing the dimension of a data set. For example, if a data set contained 150 different variables, PCA would output 150 different components, but the first couple of components would have the vast majority of the variation. This allows researchers and data scientists to reduce the time and memory it takes to run an analysis on an entire data set. They can instead use a few selected columns of PCA scores to obtain similar results in a reasonable time frame. In the case of our audio data, we have 90 variables that contain data, and more importantly, some amount of variation within each variable. This variation helped our two regression models differentiate and predict year and if a song was released before or after 1996, respectively. By running the function **prcomp()$sd** on our audio data, we can determine what percentage of variation is represented in each PCA component, as shown in the figure below.

```
 [1] 0.1408190874 0.0781407500 0.0622271777 0.0457040704 0.0361795319 0.0309734413 0.0274689882 0.0262455711 0.0255964002
[10] 0.0231300859 0.0220827432 0.0203303098 0.0189580837 0.0174471967 0.0172999986 0.0166817381 0.0155692264 0.0152460445
[19] 0.0147928404 0.0130309238 0.0128622678 0.0126716479 0.0123994319 0.0120259761 0.0117142823 0.0113500365 0.0105557676
[28] 0.0101780178 0.0096756214 0.0092961898 0.0089529077 0.0083069090 0.0081606728 0.0076380757 0.0075278659 0.0073142674
[37] 0.0072349199 0.0070728348 0.0070224701 0.0067228836 0.0066853609 0.0065063250 0.0062332914 0.0059198251 0.0058589354
[46] 0.0057386298 0.0055349530 0.0053261900 0.0052573262 0.0050109993 0.0048419438 0.0044816502 0.0042505104 0.0042286647
[55] 0.0041194488 0.0040272381 0.0038197743 0.0035749408 0.0034532519 0.0033725311 0.0031944868 0.0031417993 0.0029453328
[64] 0.0026616558 0.0026262923 0.0025629083 0.0025075682 0.0024345310 0.0022756437 0.0019911142 0.0019678950 0.0018955667
[73] 0.0017403098 0.0016553250 0.0014941830 0.0013504245 0.0013178670 0.0012352894 0.0009562757 0.0009325916 0.0007267553
[82] 0.0006498591 0.0005970572 0.0005175878 0.0004645225 0.0004193373 0.0003294046 0.0002303767 0.0002025897 0.0001264042
```

Figure 6: Percentage of Variation Represented in Each PCA Component

Likewise, similar to mean squared prediction error, mean squared error is a method of estimating the variation between fitted values and observed values. A low MSE would

indicate that the model has a good fit, while a high MSE would indicate that the model has significant room for adjustment. The MSE of a model must be compared to MSEs of other models to conclude whether the fit is sufficient. In this case, we have the MSE values of 90 different models to compare below.

```
    [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]     [,8]     [,9]    [,10]    [,11]    [,12]    [,13]
118.5442 118.2771 117.6057 117.5811 115.6603 115.1804 115.179 115.1641 115.1449 114.8564 114.7603 114.3937 113.9386
   [,14]    [,15]    [,16]   [,17]    [,18]   [,19]    [,20]    [,21]   [,22]    [,23]    [,24]    [,25]    [,26]
112.9275 112.9171 112.8309 112.442 112.3565 112.184 112.1066 112.1076 112.037 111.9379 110.9606 110.4771 110.4615
   [,27]   [,28]   [,29]    [,30]    [,31]    [,32]    [,33]    [,34]    [,35]    [,36]    [,37]    [,38]    [,39]
110.4627 110.401 110.175 110.1383 110.1287 109.9593 109.9557 109.9333 109.9292 109.9275 109.8557 109.8395 109.6854
   [,40]   [,41]   [,42]    [,43]    [,44]    [,45]    [,46]    [,47]   [,48]    [,49]    [,50]    [,51]    [,52]
109.6356 109.345 109.143 108.7753 108.7002 108.4966 108.4926 108.4421 108.436 108.2932 108.0519 108.0338 108.0352
   [,53]    [,54]    [,55]    [,56]    [,57]    [,58]    [,59]    [,60]    [,61]    [,62]    [,63]    [,64]    [,65]
108.0218 107.9629 107.9606 107.8957 107.8204 107.8233 107.8228 107.6839 107.6324 107.6332 107.6276 107.5163 107.5165
   [,66]    [,67]    [,68]   [,69]    [,70]  [,71]    [,72]    [,73]    [,74]    [,75]    [,76]    [,77]    [,78]
107.5091 107.4015 107.3234 106.917 106.8938 106.84 105.4218 105.3734 105.3749 104.7464 104.7187 104.7148 104.6982
   [,79]    [,80]    [,81]    [,82]    [,83]    [,84]    [,85]   [,86]   [,87]  [,88]    [,89]    [,90]
104.6286 103.9495 103.5529 103.4619 102.4951 101.8761 101.4251 99.5875 98.9223 92.721 91.10164 90.89855
```

Figure 7: MSE by Number of PCA Components

The question posed has a very straightforward answer; anyone who knows about PCA could surmise that the MSE would decrease by adding more PCA components, just as was reasoned analogously in Sec. 3.2.4. However, proving that theory based on our data is a difficult endeavor as it is very computationally intense. Figuring out how to configure the for loop (3 lines which can be seen in the appendix below) took many hours, and crashed many computers and RStudio sessions. We obtained each PC component by once using the function **prcomp()$x**, which outputted all of the PCA scores for each element in the original data set. We then ran a for loop that constructed the linear regression model 90 times, while adding a PCA component each time. The loop also inputted the MSE of that particular model into a data frame. In the end, we plotted the data frame and proved our theory through the figure below.

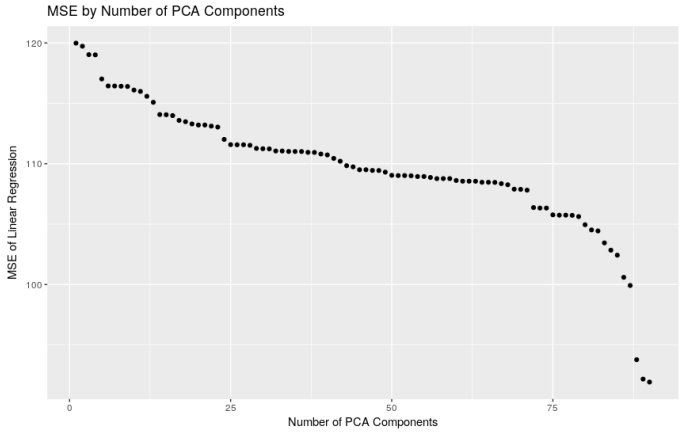


Figure 8: Plot of MSE by Number of PC Components

We can conclude that while the MSE does decrease by adding more PCA components, the decrease of adding a component is quite minimal for most components. This is because

the variation is spread out between the PCA components relatively evenly. Since the first PCA component only accounts for 14% of the total variation, the component by itself would not be a good predictor for the model. Adding subsequent components (the equivalent of more predictors) would solve this problem. The question then becomes how many components it would take to develop a good predictive model. For example, taking 10 PCA components would account for 50% of the variation in the original data set, but the remaining 50% would not be taken into account. In order to account for 90% of the variation, one would have to use 48 PCA components. A statistical analysis run using 48 of our 90 total PCA components would result in a similar conclusion to one using all 90 components, and could theoretically cut the time and memory usage of that computationally intensive analysis in half.

### 3.3 Conclusion

When running both the linear regression and PCA analyses, it grew increasingly clear that the two methods had several things in common. Developing a predictive model with enough predictors to yield fairly accurate outcomes and yet have fewer parameters than the full model to reduce complexity is a goal of both methods. In our case, there were a few, if not a dozen variables of the 90 that contributed more to the overall predictive power than the others (some were pointed out explicitly above). When keeping all predictors in the linear regression model, we noticed that the mean squared prediction error was the same as the mean squared error when all principal components remained in the model in Sec. 3.2.6, which makes intuitive sense since both methods use the same foundational formula - comparing fitted/predicted values and observed values. We also noticed that the logistic regression model does a fair job of predicting if songs were released before or after 1996; the classification accuracy was about 78.68%, suggesting that it could be possible that Autotuning changed at least some of the numeric values in the audio variables significantly. In short, the regression methods and PCA presented above hold a great amount of weight in the efforts to analyze predictive power in any study, and if the amount of data provided is large enough, stratifying the data into two distinct sets can be a useful tactic in developing strong models.

# 4 Works Cited

## References


[1] Analytics Vidhya. *data.table() vs. data.frame() - Learn to work on large data sets in R.* https://www.analyticsvidhya.com/blog/2016/05/data-table-data-frame-work-large-data-sets/. Web. Access: 2017 March 23.

[2] Cyanogen Imaging Maxlm DL. *Low-Pass Filtering (Blurring).* https://diffractionlimited.com/help/maximdl/Low-Pass_Filtering.htm. Web. Access: 2017 March 22.

[3] Gillespie, C. *Importing Data.* https://csgillespie.github.io/efficientR/5-3-importing-data.html. Web. Access: 2017 March 23.

[4] Goshtasby, et al. *An Adaptive Window Mechanism for Image Smoothing.* Web. Access: 2017 March 22.

[5] read.table package FAQ. *FAQs about the read.table package in R* http://datatable.r-forge.r-project.org/datatable-faq.pdf. Web. Access: 2017 March 23.

[6] Ross, N. *Vectorization in R - Why?* 2014 April 16. http://www.noamross.net/blog/2014/4/16/vectorization-in-r--why.html. Web. Access: 2017 March 23.

**Problem A Code:**

```
# ECS 132 Final Project: Problem A

library(pixmap) # package required
setwd('.../ECS132/TermProject') # set appr. w.d.
lll <- read.pnm('LLL.pgm') # read in the image
plot(lll)
# returns an R list consisting of a vector y and a matrix x:
# vector y: all the values of the interior pixels,
# in column-major order - length (nr-2)*(nc-2)
# matrix x: (nr-2)*(nc-2) rows and 4 columns,
# for the N/S/E/W neighbor pixel values
getnonbound = function(imgobj){
  all = imgobj@grey # extract the full pixel array,
  # numbers in [0,1], darkest to lightest
  nr = dim(imgobj@grey)[1] # number of rows in full pixel array
  nc = dim(imgobj@grey)[2] # number of columns in full pixel array
  interior = all
  interior[c(1,nr),] = 2 # assign first and last rows ("exterior rows") to 2
  interior[,c(1,nc)] = 2 # assign first and last cols ("exterior rows") to 2
  y = interior[2:(nr-1), 2:(nc-1)] # y is assigned the matrix
  # that is truly the "interior" region
  y = as.vector(y) # convert to vector (defaults to column-major order)
  x = matrix(, nrow = length(y), ncol = 4) # empty matrix,
  # dimensions of interior region and 4
  count = 0
  for(j in 1:ncol(all)){
    for(i in 1:nrow(all)){
      if(interior[i,j] != 2){ # for all entries in the true interior region...
        count = count + 1
        # assign empty matrix N, S, E, W
        x[count,] = c(all[i-1,j], all[i+1,j], all[i,j+1], all[i,j-1])
      }
    }
  }
  return (list(y,x)) # returned list requested
}

# make noise picutre
lll_noise = lll
# get the pixel values
lll_grey = lll@grey
```

```
# randomly sample and replace pixel values with values between (0,1);
# can choose p between 0 and 1
lll_grey[sample(1:length(lll_grey), length(lll_grey)*.2,
                replace = FALSE )] = runif(length(lll_grey)*0.2, 0, 1)
# reset pixel values
lll_noise@grey = lll_grey
# plot noisy image
plot(lll_noise)

# Makes a new file called denoise.pgm, in the current working directory
# the new file uses values of the NSEW pixels
# to predict the value of all pixels
# in an effort to de-noise the inputted image.
denoise = function(imgname){
  nr = dim(imgname@grey)[1] # number of rows in full pixel array
  nc = dim(imgname@grey)[2] # number of columns in full pixel array
  xy = getnonbound(imgname) # use function nonbound to get x, y
  y = xy[1][[1]]
  x = xy[2][[1]]
  # get coefficients of x and y
  pred = coef(summary(lm(y~x[,1]+x[,2]+x[,3]+x[,4])))[2:5,1]
  # multiply the coefficients by the NSEW values
  pic = as.vector(x %*% pred)
  # make sure no values are above 1 or under 0
  for(i in 1:length(pic)){
    if (pic[i] > 1)
      pic[i] = 1
    if (pic[i] < 0)
      pic[i] = 0
  }
  # make image into a matrix
  pic = matrix(data = pic, nrow = nr - 2,
               ncol = nc - 2)
  # set new interior values based on predicted values
  imgname@grey[2:(nr-1), 2:(nc-1)] = pic
  plot(imgname)
  # write to file
  write.pnm(imgname, 'denoised.pgm')
}

denoise(lll_noise) # call function
```

**Problem B Code:**

```
# ECS 132 Final Project: Problem B
setwd('/home/ckashyap/Downloads') # set appr. w.d.

# http://stackoverflow.com/questions/6262203/
# measuring-function-execution-time-in-r
# get original time
time_csv = Sys.time()
# read in file
YearPredictionMSD <- read.csv('YearPredictionMSD.txt',
                              header = FALSE, sep = ",")
time_csv = Sys.time() - time_csv
# get time difference
time_csv

# load in library
library(data.table)
# get original time
time_fread = Sys.time()
# read in file
YearPredictionMSD <- fread("YearPredictionMSD.txt")
time_fread = Sys.time() - time_fread
# get time difference
time_fread

# YearPredictionMSD = YearPredictionMSD[sample(nrow(YearPredictionMSD),
        # ceiling(length(YearPredictionMSD$V1)*(.1))),]

# randomly sample for index numbers to differentiate between test/training data
index_numbers = sample(nrow(YearPredictionMSD),
                       ceiling(length(YearPredictionMSD$V1)*(1/3)))
test = YearPredictionMSD[index_numbers,]
training = YearPredictionMSD[!index_numbers,]

# make two dfs which differentiate if song was before or after 1996
train_today = training[which(training$V1 >= 1996)]
train_1996 = training[which(training$V1 < 1996)]

# get mean of V77 of both dfs
old77 = mean(train_1996$V77)
new77 = mean(train_today$V77)

```

```
# fit a regression model with all variables V2 to V91
train_fit = lm(V1 ~ ., data = training)
summary(train_fit)
# predict year using the fit of our model
test_predict = predict(train_fit, test, interval = 'prediction')[,1]
# get average squared sum error or MSE
avgSE = sum((test$V1 - test_predict)^2)/length(test_predict)

# create binary variable for coming logistic regression
# in both test and training
training$autotune = 1
training$autotune[which(training$V1 >= 1996)] = 2
training$autotune = as.factor(training$autotune)
test$autotune = 1
test$autotune[which(test$V1 >= 1996)] = 2
test$autotune = as.factor(test$autotune)

# fit a logit model with all variables V2 to V91
log_fit = glm(autotune ~ . - V1, data = training, family = binomial)
summary(log_fit)
# predict using response as type, so predictions are between 0 and 1
log_predict = predict(log_fit, test, type = 'response')
# now we can view as predictions, 1 or 2
log_predict = (log_predict > 0.5) + 1
# get table of predictions based on if prediction was right or not
log_con = table(true = test$autotune, model = log_predict)
# get error rate using table
error_rate = (log_con[2] + log_con[3])/length(test$V1)

# delete binary variable
training = training[,-92]
test = test[,-92]

# run pca on V2 to V91 and get scores for every element
pca = data.frame(prcomp(YearPredictionMSD[,-1])$x)
# append V1 onto dataframe
pca = data.frame(append(pca, YearPredictionMSD[,1]))
# create test/training using previous index numbers
pca_test = data.frame(pca[index_numbers,])
pca_training = data.frame(pca[-index_numbers,])
#what percentage of variation is represented in each component
pca_var = prcomp(YearPredictionMSD[,-1])$sd
```

```
84 pca_var = pca_var/sum(pca_var)
85
86 # create storage for the MSE
87 pcaMSE = numeric(0)
88
89 # loop to run linear regression
90 #http://tex.stackexchange.com/questions/11177/how-to-write-hidden-notes-in-a-lat
91 for(i in 1:90){
92   # fit formula everytime and paste formula needed
93   fit_pca =lm(as.formula(c("V1 ~",
94                            paste(names(pca_training)[1:i], collapse = "+"))),
95                            data = pca_training)
96   # predict year using the fit of our model on the test df
97   test_predict = predict(fit_pca, pca_test, interval = 'prediction')[,1]
98   # calculate the MSE of our model
99   pcaMSE = c(pcaMSE, sum((pca_test$V1 - test_predict)^2)/length(test_predict))
100 }
101
102 # load in library
103 library(ggplot2)
104 # create df, since ggplot doesn't use vectors well
105 pcaMSE = data.frame(pcaMSE)
106 # plot our MSE by Number of PCA components
107 ggplot(data = pcaMSE) + geom_point(aes(x=seq(1:90), y=pcaMSE)) +
108    ggtitle("MSE by Number of PCA Components") +
109    labs(x = 'Number of PCA Components', y = 'MSE of Linear Regression')
110
111 sum(pca_var[1:10])
112 sum(pca_var[1:48])
```

# A Contributions

Chad Pickering: Majority of report. Helped with code.
Chirag Kashyap: Majority of code. Helped with report.