A decade or more ago a nice worked example from the political scientist Simon Jackman demonstrating how to do [Principal Components Analysis](https://en.wikipedia.org/wiki/Principal_component_analysis). PCA is one of the basic techniques for reducing data with multiple dimensions to some much smaller subset that nevertheless represents or condenses the information we have in a useful way. In a PCA approach, we transform the data in order to find the “best” set of underlying components. We want the dimensions we choose to be orthogonal to one another—that is, linearly uncorrelated. PCA is an inductive approach to data analysis. Because of the way it works, we’re arithmetically guaranteed to find a set of components that “explain” all the variance we observe. The *substantive* explanatory question is whether the main components uncovered by PCA have a plausible interpretation.

It was reminded of all of this on Friday because some of my first-year undergrad students are doing an “Algorithms for Data Science” course, and the topic of PCA came up there. Some students not in that class wanted some intuitions about what PCA was. The thing I remembered about Jackman’s discussion was that he had the nice idea of doing PCA on an image, in order to show both how you could reconstruct the whole image from the PCA, if you wanted, and more importantly to provide some intuition about what the first few components of a PCA picked up on. His discussion doesn’t seem to be available anymore, This one:



Elvis Meets Nixon.

**Setup**

The [Imager Library](https://dahtah.github.io/imager/imager.html) is our friend here. It’s a great toolkit for processing images in R, and it’s friendly to tidyverse packages, too.

library(imager)

library(here)

library(dplyr)

library(broom)

library(ggplot2)

**Load the image**

Our image is in a subfolder of our project directory. The load.image() function is from Imager, and imports the image as a cimg object. The library provides a method to convert these objects to a long-form data frame. Our image is greyscale, which makes it easier to work with. It’s 800 pixels wide by 633 pixels tall.

img <- load.image(here("img/elvis-nixon.jpeg"))

str(img)

## 'cimg' num [1:800, 1:633, 1, 1] 0.914 0.929 0.91 0.906 0.898 ...

dim(img)

## [1] 800 633 1 1

img\_df\_long <- as.data.frame(img)

head(img\_df\_long)

## x y value

## 1 1 1 0.9137255

## 2 2 1 0.9294118

## 3 3 1 0.9098039

## 4 4 1 0.9058824

## 5 5 1 0.8980392

## 6 6 1 0.8862745

Each x-y pair is a location in the 800 by 633 pixel grid, and the value is a grayscale value ranging from zero to one. To do a PCA we will need a matrix of data in wide format, though—one that reproduces the shape of the image, a row-and-column grid of pixels, each with some a level of gray. We’ll widen it using pivot\_wider:

img\_df <- tidyr::pivot\_wider(img\_df\_long,

names\_from = y,

values\_from = value)

dim(img\_df)

## [1] 800 634

So now it’s the right shape. Here are the first few rows and columns.

img\_df[1:5, 1:5]

## # A tibble: 5 x 5

## x `1` `2` `3` `4`

##

## 1 1 0.914 0.914 0.914 0.910

## 2 2 0.929 0.929 0.925 0.918

## 3 3 0.910 0.910 0.902 0.894

## 4 4 0.906 0.902 0.898 0.894

## 5 5 0.898 0.894 0.890 0.886

The values stretch off in both directions. Notice the x column there, which names the rows. We’ll drop that when we do the PCA.

**Do the PCA**

Next, we do the PCA, dropping the x column and feeding the 800×633 matrix to Base R’s prcomp() function.

img\_pca <- img\_df %>%

dplyr::select(-x) %>%

prcomp(scale = TRUE, center = TRUE)

There are a lot of components—633 of them altogether, in fact, so I’m only going to show the first twelve and the last six here. You can see that by component 12 we’re already up to almost 87% of the total variance “explained”.

summary(img\_pca)

## Importance of components:

## PC1 PC2 PC3 PC4 PC5 PC6

## Standard deviation 15.2124 10.9823 7.54308 5.57239 4.77759 4.55531

## Proportion of Variance 0.3656 0.1905 0.08989 0.04905 0.03606 0.03278

## Cumulative Proportion 0.3656 0.5561 0.64601 0.69506 0.73112 0.76391

## PC7 PC8 PC9 PC10 PC11 PC12

## Standard deviation 4.0649 3.66116 3.36891 3.27698 2.82984 2.49643

## Proportion of Variance 0.0261 0.02118 0.01793 0.01696 0.01265 0.00985

## Cumulative Proportion 0.7900 0.81118 0.82911 0.84608 0.85873 0.86857

## [A lot more components]

## PC628 PC629 PC630 PC631 PC632

## Standard deviation 0.001125 0.001104 0.001097 0.001037 0.000993

## Proportion of Variance 0.000000 0.000000 0.000000 0.000000 0.000000

## Cumulative Proportion 1.000000 1.000000 1.000000 1.000000 1.000000

## PC633

## Standard deviation 0.0009215

## Proportion of Variance 0.0000000

## Cumulative Proportion 1.0000000

We can tidy the output of prcomp with broom’s tidy function, just to get a summary scree plot showing the variance “explained” by each component.

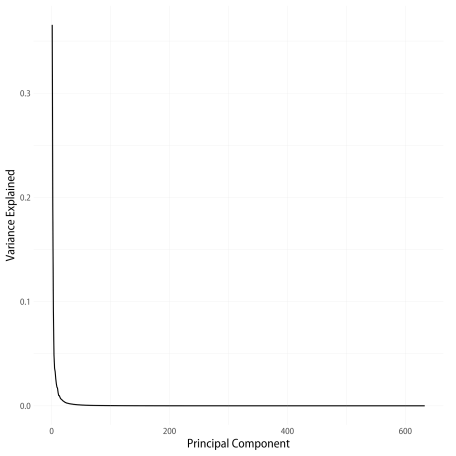
pca\_tidy <- tidy(img\_pca, matrix = "pcs")

pca\_tidy %>%

ggplot(aes(x = PC, y = percent)) +

geom\_line() +

labs(x = "Principal Component", y = "Variance Explained")



Scree plot of the PCA.

**Reversing the PCA**

Now the fun bit. The object produced by prcomp() has a few pieces inside:

names(img\_pca)

## [1] "sdev" "rotation" "center" "scale" "x"

What are these? sdev contains the standard deviations of the principal components. rotation is a matrix where the rows correspond to the columns of the original data, and the columns are the principal components. x is a matrix containing the value of the rotated data multiplied by the rotation matrix. Finally, center and scale are vectors with the centering and scaling information for each observation.

Now, to get from this information back to the original data matrix, we need to multiply x by the transpose of the rotation matrix, and then revert the centering and scaling steps. If we multiply by the transpose of the *full* rotation matrix (and then un-center and un-scale), we’ll recover the original data matrix exactly. But we can also choose to use just the first few principal components, instead. There are 633 components in all (corresponding to the number of rows in the original data matrix), but the scree plot suggests that most of the data is “explained” by a much smaller number of components than that.

Here’s a function that takes a PCA object created by prcomp() and returns an approximation of the original data, calculated by some number (n\_comp) of principal components. It returns its results in long format, in a way that mirrors what the Imager library wants. This will make plotting easier in a minute.

reverse\_pca <- function(n\_comp = 20, pca\_object = img\_pca){

## The pca\_object is an object created by base R's prcomp() function.

## Multiply the matrix of rotated data by the transpose of the matrix

## of eigenvalues (i.e. the component loadings) to get back to a

## matrix of original data values

recon <- pca\_object$x[, 1:n\_comp] %\*% t(pca\_object$rotation[, 1:n\_comp])

## Reverse any scaling and centering that was done by prcomp()

if(all(pca\_object$scale != FALSE)){

## Rescale by the reciprocal of the scaling factor, i.e. back to

## original range.

recon <- scale(recon, center = FALSE, scale = 1/pca\_object$scale)

}

if(all(pca\_object$center != FALSE)){

## Remove any mean centering by adding the subtracted mean back in

recon <- scale(recon, scale = FALSE, center = -1 \* pca\_object$center)

}

## Make it a data frame that we can easily pivot to long format

## (because that's the format that the excellent imager library wants

## when drawing image plots with ggplot)

recon\_df <- data.frame(cbind(1:nrow(recon), recon))

colnames(recon\_df) <- c("x", 1:(ncol(recon\_df)-1))

## Return the data to long form

recon\_df\_long <- recon\_df %>%

tidyr::pivot\_longer(cols = -x,

names\_to = "y",

values\_to = "value") %>%

mutate(y = as.numeric(y)) %>%

arrange(y) %>%

as.data.frame()

recon\_df\_long

}

Let’s put the function to work by mapping it to our PCA object, and reconstructing our image based on the first 2, 3, 4, 5, 10, 20, 50, and 100 principal components.

## The sequence of PCA components we want

n\_pcs <- c(2:5, 10, 20, 50, 100)

names(n\_pcs) <- paste("First", n\_pcs, "Components", sep = "\_")

## map reverse\_pca()

recovered\_imgs <- map\_dfr(n\_pcs,

reverse\_pca,

.id = "pcs") %>%

mutate(pcs = stringr::str\_replace\_all(pcs, "\_", " "),

pcs = factor(pcs, levels = unique(pcs), ordered = TRUE))

This gives us a very long tibble with an index (pcs) for the number of components used to reconstruct the image. In essence it’s eight images stacked on top of one another. Each image has been reconstituted using a some number of components, from a very small number (2) to a larger number (100). Now we can plot each resulting image in a small multiple.

p <- ggplot(data = recovered\_imgs,

mapping = aes(x = x, y = y, fill = value))

p\_out <- p + geom\_raster() +

scale\_y\_reverse() +

scale\_fill\_gradient(low = "black", high = "white") +

facet\_wrap(~ pcs, ncol = 2) +

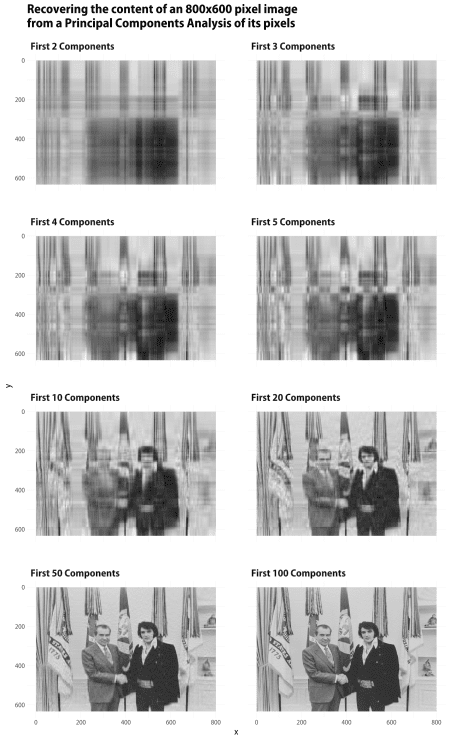
guides(fill = FALSE) +

labs(title = "Recovering the content of an 800x600 pixel image\nfrom a Principal Components Analysis of its pixels") +

theme(strip.text = element\_text(face = "bold", size = rel(1.2)),

plot.title = element\_text(size = rel(1.5)))

p\_out



Elvis Meets Nixon, as recaptured by varying numbers of principal components.

There’s a lot more one could do with this, especially if I knew rather more linear algebra than I in fact do haha. But at any rate we can see that it’s pretty straightforward to use R to play around with PCA and images in a tidy framework.